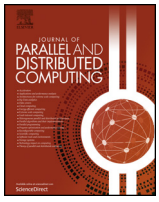
Journal of Parallel and Distributed Computing 183 (2024) 104767

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



SUARA: A scalable universal allreduce communication algorithm for acceleration of parallel deep learning applications ☆

Emin Nuriyev [a], Ravi Reddy Manumachu [a,*], Samar Aseeri [b], Mahendra K. Verma [c], Alexey L. Lastovetsky [a]

[a] School of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland
[b] Extreme Computing Research Center (ECRC), King Abdullah University of Science and Technology, Thuwal, Saudi Arabia
[c] Department of Physics, Indian Institute of Technology, Kalyanpur, Kanpur, India

ARTICLE INFO

Article history:
Received 1 March 2023
Received in revised form 31 July 2023
Accepted 4 September 2023
Available online 15 September 2023

Keywords:
Allreduce communication algorithm
MPI
Parallel deep learning
ResNet-50
Imagenet

ABSTRACT

Parallel and distributed deep learning (PDNN) has become an effective strategy to reduce the long training times of large-scale deep neural networks. Mainstream PDNN software packages based on the message-passing interface (MPI) and employing synchronous stochastic gradient descent rely crucially on the performance of MPI allreduce collective communication routine.

In this work, we propose a novel scalable universal allreduce meta-algorithm called SUARA. In general, SUARA consists of L serial steps, where $L \geq 2$, executed by all MPI processes involved in the allreduce operation. At each step, SUARA partitions this set of processes into subsets, which execute optimally selected library allreduce algorithms to solve sub-allreduce problems on these subsets in parallel, to accomplish the whole allreduce operation after completing all the L steps. We then design, theoretically study and implement a two-step SUARA ($L = 2$) called SUARA2 on top of the Open MPI library. We prove that the theoretical asymptotic speedup of SUARA2 executed by P processes over the base Open MPI routine is $\mathcal{O}(\sqrt{P})$. Our experiments on Shaheen-II supercomputer employing 1024 nodes demonstrate over 2x speedup of SUARA2 over native Open MPI allreduce routine, which translates into the performance improvement of training of ResNet-50 DNN on ImageNet by 9%.

1. Introduction

Deep learning (DL) applications have become pervasive energizing technological innovations in several fields that include speech recognition [3], autonomous driving [6], medical diagnosis [12], and natural language processing [13].

Complex DL applications require training deep neural networks (DNNs) on large datasets for better predictions. However, the training times increase drastically with the size of DNN given by the number of parameters and the size of the training dataset. Therefore, parallel and distributed DL (PDNN) has become a natural and effective strategy to reduce the long training times of large-scale DNNs.

Horovod [39], Microsoft Cognitive Toolkit (CNTK) [38], and MXNet MPI [28] are popular PDNN packages that perform parallel training of a DNN using data-parallelism and synchronous model updates. In the data-parallel approach, the training dataset of samples is divided into small batches called mini-batches. The set of mini-batches is then partitioned equally between the processes. The complete training process typically consists of hundreds of epochs. An epoch comprises a loop where each process selects a disjoint mini-batch in an iteration. The process then executes the DNN code using this batch and computes a gradient. All the processes then collectively invoke an *allreduce collective communication operation* to obtain the global average gradient for the whole minibatch. Finally, each process then updates the local vector of weights using the global average gradient. The main stages of parallel training of a DNN are detailed in supplemental, Section 2.

Therefore, the allreduce collective communication routine is an essential ingredient of PDNN packages that employ data parallelism and synchronous model updates [39], [38], [28].

☆ This publication has emanated from research conducted with the financial support of Science Foundation Ireland and the Sustainable Energy Authority of Ireland under the SFI Frontiers for the Future Programme 20/FFP-P/8683. This publication has emanated from research conducted with the financial support of Sustainable Energy Authority of Ireland (SEAI) under Grant Number 21/RDD/664.

* Corresponding author.
E-mail addresses: nuriyevemin@gmail.com (E. Nuriyev), ravi.manumachu@ucd.ie (R.R. Manumachu), samar.aseeri@kaust.edu.sa (S. Aseeri), mkv@iitk.ac.in (M.K. Verma), alexey.lastovetsky@ucd.ie (A.L. Lastovetsky).

https://doi.org/10.1016/j.jpdc.2023.104767

The MPI standard [29], which provides a reliable and portable environment for developing HPC applications, offers a rich set of collective communication operations, including the allreduce collective communication operation. Different algorithms have been developed and implemented for the allreduce MPI collective operation, but no algorithm proved superior in all situations. Therefore, MPI implementations must solve the problem of selecting the optimal algorithm for the collective operation depending on the platform, the number of processes involved, and the message size. The Open MPI library [18] supports runtime selection of six different algorithms for the MPI allreduce collective communication operation, namely, *linear* (*linear reduce* followed by *linear broadcast*) [34], *nonoverlapping* (*tuned reduce* followed by *tuned broadcast*) [34], *recursive doubling* [34], *ring* [34], *ring with segmentation* [35], *Rabenseifner* [35].

Methods for performance optimization of the allreduce collective communication can be broadly classified into *platform-specific* and *platform-independent* categories. Platform-specific methods aim to optimize the allreduce for performance for a specific platform [42], [44], [20], [41], [27], [2], [4], [24]. In the *platform-independent* category, research works include algorithms that do not make any assumptions about the underlying platform [35], [9], [33], [46], [7], [40].

The *platform-independent* category can be further classified into two sub-categories. The first sub-category comprises research works [35], [9], [33], [40] employing *functional decomposition* of the global allreduce operation into a serial sequence of collective sub-operations different from allreduce. The second sub-category contains research works [46], [7] that employ *message decomposition/segmentation/pipelining*.

In this work, we propose a novel scalable universal allreduce meta-algorithm called *SUARA*. In general, SUARA consists of several serial steps executed by all processes involved in the allreduce operation. The processes contain messages of the same length for reduction. At each step, SUARA partitions the whole set of processes into subsets, which execute *allreduce* algorithms, optimally selected from a given set of allreduce algorithms, A, to solve sub-allreduce problems on these subsets in parallel, accomplishing the whole allreduce operation after the completion of all the serial steps. Furthermore, it does not use message decomposition; therefore, the sub-allreduce operations compute partial reductions of the whole message.

SUARA is a meta-algorithm since it represents a family of algorithms, parameterized by the number of serial steps, L, and the set A of native allreduce algorithms used as building blocks. There are no restrictions on the native allreduce algorithms (for example, they can use either functional or message decomposition or both in their execution).

Thus, SUARA is a platform-independent multi-step allreduce meta-algorithm employing *process decomposition* to optimize the global allreduce operation using only *native allreduce algorithms* as sub-operations. It differs from the state-of-the-art platform-independent allreduce algorithms in two respects. First, it is based on process decomposition, not functional or message decomposition. Second, it only employs sub-allreduce operations executing optimally selected native allreduce algorithms as its building blocks.

We first prove that the processes executing SUARA must naturally form an L-dimensional rectangular arrangement for maximum parallelism and to ensure the correctness of the allreduce operation.

We then design, theoretically study and implement a two-step SUARA ($L = 2$), called *SUARA2*, on top of the Open MPI library. The processes in SUARA2 form a two-dimensional grid arrangement. The design and implementation of SUARA2 comprise three stages. At the first stage, SUARA2 determines the optimal 2D process grid arrangement and the optimal Open MPI allreduce algorithms to employ in the process rows and columns. In the second stage, process rows execute library allreduce algorithms in parallel. At the third stage, process columns execute library allreduce algorithms in parallel, completing the whole allreduce operation. SUARA2 automatically selects optimal library allreduce algorithms to be executed by process rows and columns from the set of algorithms implemented by Open MPI. We prove that the optimal selection always uses at most *two* different library algorithms – one for all process rows and the other for all process columns. We also prove that the theoretical asymptotic speedup of SUARA2 executed by a set of P processes over the best Open MPI allreduce algorithm is $\mathcal{O}(\sqrt{P})$.

Our goal of the paper is not to develop an optimal allreduce algorithm from the total space of allreduce algorithms that employ message segmentation/pipelining, functional decomposition into allreduce and non-allreduce collective operations, and process decomposition. This endeavour is out of the scope of this work. Instead, we focus on finding the optimal allreduce algorithm in the space of allreduce algorithms employing process decomposition.

We demonstrate the practical efficiency of SUARA2 by speeding up ResNet-50 DNN training on ImageNet dataset [37] on Shaheen-II supercomputer employing 1024 dual-socket 16-core Intel Haswell processors [26]. We focus only on one-process-per-node application configuration. Other pertinent application configurations that include one-process-per-socket and one-process-per-core are out of the scope of this work.

The PDNN framework used is Horovod [39] employing Open MPI library 4.0.3 for communication. The main stages of parallel training of a DNN are detailed in supplemental, Section 2. Each process passes a message, a vector of gradients of size m bytes, to MPI_Allreduce collective routine invoked during Resnet-50 DNN training. All the processes call the MPI_Allreduce collective routine during Step 3 of an epoch to obtain the same global vector of average gradients from the input vectors of gradients.

The reduction of training time due to using SUARA2 increases with the number of employed processes. It reaches 9% for 1024 processes, the maximal number used in the experiments. The minimum, average, and maximum speedups of SUARA2 over the best native Open MPI allreduce routine observed in our experiments are 1.6x, 2x, and 2.65x, respectively.

The main contributions of this work are:

- A platform-independent multi-step scalable universal allreduce meta-algorithm called SUARA that employs the novel approach of process decomposition to optimize the global allreduce operation using only native allreduce algorithms as sub-operations;
- A detailed design and theoretical analysis of a two-step SUARA, called SUARA2, on top of the Open MPI set of allreduce algorithms. SUARA2 exhibits a theoretical asymptotic speedup of $\mathcal{O}(\sqrt{P})$ over the best Open MPI allreduce algorithm;
- An Open MPI library-based portable implementation of SUARA2;
- Using traditional calculus approach to determine the optimal allreduce combination (allreduce algorithms in rows and columns) to employ during the execution of the MPI allreduce collective operation;
- Experimental demonstration of the practical efficiency of SUARA2 in PDNN by the 9% acceleration of the training of ResNet-50 DNN on ImageNet dataset on Shaheen-II supercomputer employing 1024 processes (1024 nodes). SUARA2 outperforms the native Open MPI allreduce routine more than twice.

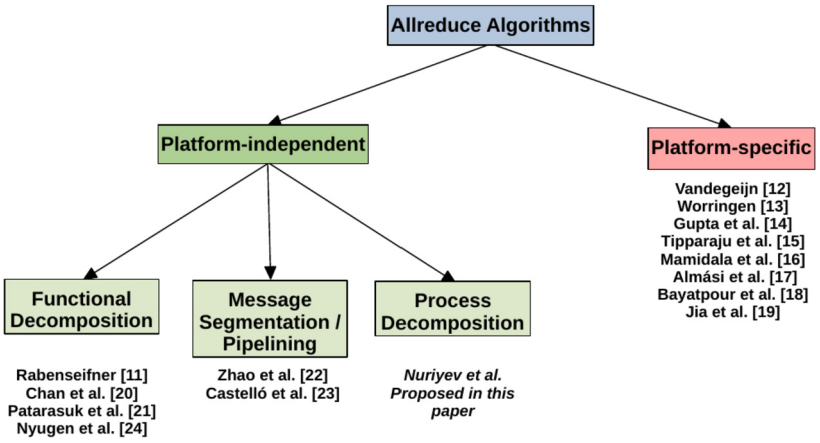


Fig. 1. Classification of allreduce collective communication algorithms.

The rest of the paper is organized as follows. The related work section reviews the existing approaches to performance optimization of collective communication operations, PDNN packages, and methods for acceleration of training of DNNs on ImageNet. We follow this with the section that presents our scalable universal allreduce meta-algorithm, SUARA. Then, we describe in detail a two-step SUARA on top of the Open MPI library. Next, the experimental results section presents the practical efficiency of SUARA2. Finally, the conclusion section ends the paper.

2. Related work

We then present an overview of the state-of-the-art methods for performance optimization of allreduce communication. We then overview DNN frameworks offering support for parallel and distributed training.

2.1. Performance optimization of allreduce collective communication

We have presented an overview of prior works in this category in the introduction section and described how our proposed allreduce meta-algorithm, SUARA, differs from these works. Therefore, we briefly cover the prior works in the *platform-independent* category here with a few additional details. Fig. 1 shows the tree ontology of different allreduce algorithms.

The *platform-independent* category can be classified into two sub-categories. The first sub-category [35], [9], [33], [40] comprises research works employing functional decomposition of the global allreduce operation into a sequence of non-allreduce sub-operations executed serially. The second sub-category contains research works [46], [7] that employ message decomposition/segmentation. This technique is also known as pipelining in the literature.

Rabenseifner et al. [35] study functional decompositions of the allreduce operation comprising two serial steps. One decomposition is a reduce operation followed by a broadcast operation, and the other involves reduce-scatter and allgather operations. They propose some known and novel algorithms for the two steps. For example, a binary tree algorithm for reduce and broadcast operations, recursive vector halving and distance doubling algorithm for reduce-scatter operation and recursive vector doubling combined with recursive distance halving algorithm for allgather operation. Finally, they experimentally find the fastest allreduce configuration (the algorithmic combination for the two steps) on a Cray supercomputer depending on the number of processes and message size.

Chan et al. [9] study specifically the MPI_Allreduce implementation in the MPICH [19] library that employs a two-step functional decomposition of the allreduce operation, reduce-scatter and allgather operations based on recursive-halving and doubling algorithms. They propose an algorithmic enhancement to the MPICH implementation that performs well for a particular range of message sizes on a Cray cluster. However, SUARA is a multi-step allreduce meta-algorithm based on process decomposition that optimizes the global allreduce operation for any input message size.

Patarasuk et al. [33] also employ a two-step functional decomposition of the allreduce operation involving reduce-scatter and allgather operations. In addition, they use logical ring-based algorithms for the reduce-scatter and allgather operations. However, SUARA is a multi-step allreduce meta-algorithm that employs process decomposition to optimize the global allreduce operation using only native allreduce algorithms as sub-operations. The allreduce algorithms for the sub-operations are optimally selected from a pool of available native algorithms that include linear, ring-based, recursive, and Rabenseifner algorithms, to name a few.

Nguyen et al. [40] present a functional decomposition of the allreduce operation comprising four serial steps for accelerating deep learning workloads on GPU clusters. The first and fourth steps involve parallel reduce-scatter operations and parallel allgather operations involving the GPUs inside each node. The second and third steps contain an inter-node allreduce operation realized by a two-step functional decomposition involving parallel reduce-scatter and allgather operations. A logical ring algorithm is used for the intra-node collective operations, whereas three different algorithms (recursive doubling, logical ring or Rabenseifner) are employed for the inter-node allreduce collective operation.

Research works [46], [7] employ message decomposition/segmentation/pipelining. Zhao et al. [46] split the message into segments that are reduced using reduce and broadcast operations, which employ pipelining. Castello et al. [7] divide the message into parts that are reduced by parallel nonblocking MPI sub-allreduce operations. However, SUARA does not use message decomposition. It computes partial reductions of the whole message using sub-allreduce operations.

2.2. Parallel and distributed machine learning packages

The mainstream PDNN packages can be classified based on the type of parallelism and the DNN model consistency. The forward evaluation and backpropagation phases of a DNN are partitioned between the processors in three different ways:

- **Data parallelism:** The work of the minibatch samples is partitioned between the processors. The results of the partitions are averaged using an allreduce collective communication operation to obtain the gradient for the whole minibatch. The

allreduce communication operation combines values from all the processes and distributes the result back to all processes.

- **Model parallelism:** The neurons in each layer are partitioned between the processors. All the processors get a copy of the minibatch and compute different parts of a DNN. Fully connected layers incur all-to-all communication compared to allreduce communication in data parallelism.
- **Layer pipelining**: There are two forms of pipelining. The first form is to overlap computations between one layer and the next. For example, forward evaluation, backpropagation, and weight updates can be overlapped. The second form is a multi-processor pipeline partitioning the DNN layers between the processors.

The surveys [5], [43] present informative descriptions of parallelism in deep learning.

In the category of DNN model consistency, the consistent model methods are based on Bulk Synchronous Parallelism (BSP), where consistency is ensured by a global synchronization step between each computation and communication phase. In BSP (or synchronous SGD), the up-to-date model parameters (vector of weights) are made visible to all the processes after each global synchronization step. Asynchronous SGD relaxes the synchronization criterion for an up-to-date weight vector and leads to an inconsistent model. A well-known instance of inconsistent SGD is HOGWILD algorithm [15], which allows processes to read model parameters and update gradients without any synchronization. Total Asynchronous Parallelism or Barrierless Asynchronous Parallelism (BAP) allows workers to communicate without synchronization. However, this technique can lead to slow convergence, and incorrectness [23]. Stale Synchronous Parallelism (SSP) offers a compromise between consistent and inconsistent models and provides strong model convergence guarantees [21]. SSP performs the global synchronization step after only one node reaches maximal staleness. This technique works well in heterogeneous environments where stragglers (lagging workers) are a vital concern.

Horovod [39] is a parallel DNN framework that employs MPI and NVIDIA Collective Communication Library (NCCL) [32] for training on CPUs or GPUs. It is based on MPI allreduce communication operation, and is therefore used in this work for demonstrating the efficiency of our library. Caffe2 [25] is also a distributed DNN framework based on allreduce. However, it uses NCCL between the GPUs on a single node and Gloo library between the nodes [16], which uses the ring and recursive doubling allreduce algorithms. The Microsoft Cognitive Tooklit (CNTK) [38] is a distributed DNN framework that represents a DNN by a directed graph and is based on ring allreduce algorithm.

The following packages are based on parallel asynchronous SGD. DistBelief [15] combines the three types of parallelism. It represents a DNN by a computation graph. The graph is partitioned between the processors using either model parallelism or pipelining. Since DistBelief provides fault tolerance, there are model replicas. The replicas are trained in parallel on different samples. Project Adam [11] also combines the three types of parallelism. DIstributed Artificial Neural NEtworks (DIANNE) [14] is a java-based distributed DNN framework employing model parallelism. Tensorflow framework [1] represents a DNN by a dataflow graph and supports data and model parallelism. MXNet [10] also represents a DNN by a dataflow graph. MXNet MPI [28] combines asynchronous (using parameter server concept) and synchronous (using MPI) implementations. The processes are divided into groups. Within each group, synchronous SGD is executed using MPI allreduce communication.

Petuum [45] supports data and model parallelism and is based on parallel stale-synchronous SGD (SSP).

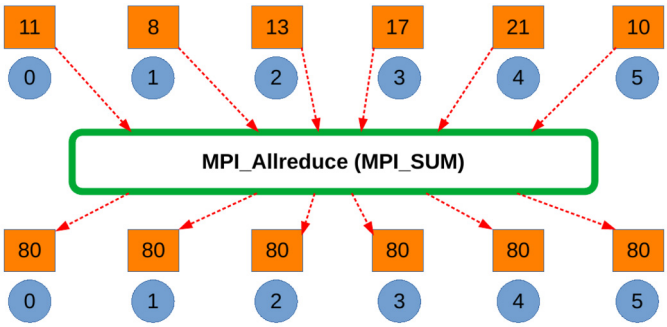


Fig. 2. Example illustrating allreduce involving six processes with ranks in the set, $S = \{0, 1, 2, 3, 4, 5\}$. A circle symbolizes a process. The local value at each process is shown in a square. At the end of the allreduce operation, all the processes will have an identical result, 80.

3. SUARA: a scalable universal allreduce collective algorithm

In this section, we propose our novel scalable universal algorithm called *SUARA* for the allreduce communication operation.

The allreduce is a collective communication operation that applies a commutative and associative operator to values from all the processes and distributes the final result to all the processes. The operators include max, min, sum, product, and logical bitwise. All the processes participating in the allreduce must have an identical result after its successful completion. The associativity and commutativity of the reduction operator signify that the values of all the processes can be rearranged and combined to determine the final result. Therefore, it allows ample scope for optimization where the whole allreduce problem can be partitioned into sub-allreduce problems that can be solved simultaneously. Fig. 2 illustrates the allreduce operation involving six processes, whose ranks are given by the set $S = \{0, 1, 2, 3, 4, 5\}$. A circle symbolizes each process. The local value at each process is shown in a square above the circle. The values are reduced using MPI_SUM operator. At the end of the allreduce operation, each process contains a result equal to 80. In this example, the local value at each process is a scalar. However, the local data can be a vector in general. Therefore, the allreduce operation reduces vectors from all the processes.

Our proposed algorithm SUARA is an allreduce meta-algorithm executed by a set S of processes of size P. The processes contain messages of the same length for reduction. SUARA consists of L serial steps, executed by all processes in S. At each step, SUARA partitions S into subsets, which execute allreduce algorithms, optimally selected from a given set of allreduce algorithms, A, to solve sub-allreduce problems on these subsets in parallel. It accomplishes the whole allreduce operation after the completion of all L steps.

SUARA is a meta-algorithm since it represents a family of algorithms, parameterized by the number of serial steps, L, and the set A of native allreduce algorithms used as its building blocks. There are no restrictions on the native allreduce algorithms.

We show that the processes of S employed in SUARA's execution must naturally form a L-dimensional rectangular arrangement to ensure its correctness and for maximum parallelism.

We first illustrate the execution of SUARA, accomplishing the whole allreduce in two steps ($L = 2$). In the first step, SUARA partitions S into R subsets. Each subset solves a sub-allreduce problem. Since the process subsets are disjoint, SUARA executes all the sub-allreduce operations in parallel. At the end of the step, all the processes in a subset contain the same result. To complete the allreduce correctly, each process must reduce its local result with the result from one process in each of the other subsets. The most natural way that maximizes the parallelism is to compose disjoint subsets where each subset is assigned a unique process from each

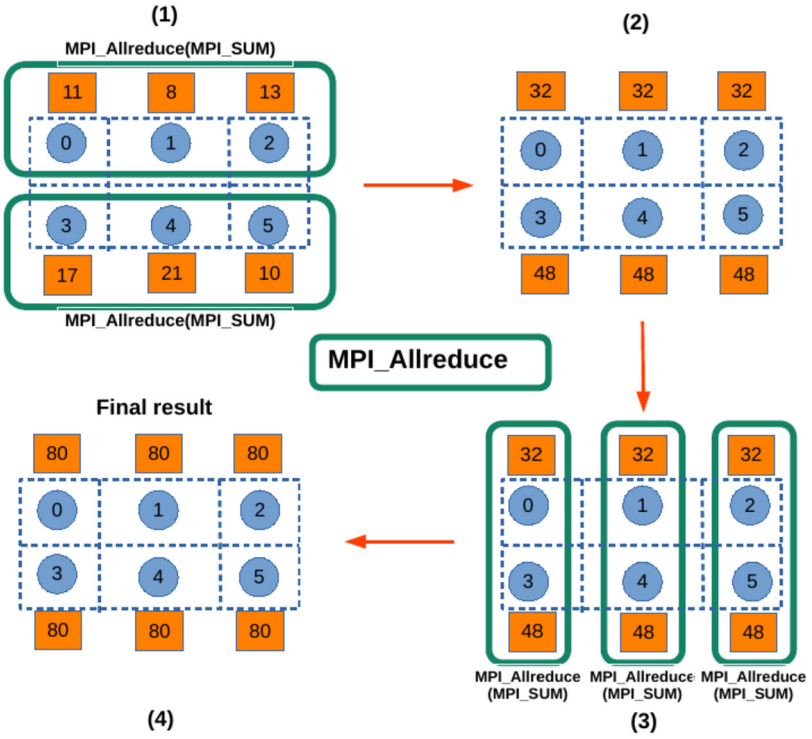


Fig. 3. Example illustrating the execution of two-step SUARA by six processes in the set, $S = \{0, 1, 2, 3, 4, 5\}$. SUARA accomplishes the whole allreduce using two steps ($L = 2$). In the first step (1), SUARA partitions S into two subsets, $\{0, 1, 2\}$ and $\{3, 4, 5\}$. It then executes two sub-allreduces in parallel on these subsets. In the second step, SUARA partitions S into three subsets, $\{0, 3\}$, $\{1, 4\}$, $\{2, 5\}$, to complete the whole allreduce correctly and executes three sub-allreduces in parallel shown in (3). Therefore, $R = 2, C = 3$. At the end of the whole allreduce operation, each process contains a result equal to 80. The final result is displayed under (4).

of the R subsets in the previous step. Therefore, SUARA composes C such disjoint subsets where $R \times C = P$. SUARA then executes all the C sub-allreduce operations in parallel. Hence, the processes of S employed in SUARA's execution must naturally form a two-dimensional grid arrangement to ensure its correctness.

Fig. 3 illustrates the execution of two-step SUARA by six processes in the set $S = \{0, 1, 2, 3, 4, 5\}$. The goal of SUARA here is to accomplish the allreduce using two steps. In the first step, SUARA partitions S into two subsets, $\{0, 1, 2\}$ and $\{3, 4, 5\}$. It then executes two sub-allreduces in parallel on these subsets. After the completion of the step, all the processes in the subset $\{0, 1, 2\}$ contain a result 32. All the processes in the subset $\{3, 4, 5\}$ contain 48. In the second step, SUARA partitions S into three subsets, $\{0, 3\}$, $\{1, 4\}$, $\{2, 5\}$, and executes three sub-allreduces in parallel. At the end of the SUARA execution, each process contains a result equal to 80. The partitions, $(\{0, 1, 2\}, \{3, 4, 5\})$ and $(\{0, 3\}, \{1, 4\}, \{2, 5\})$, form a 2D process arrangement, 2×3.

Note. In this work, we do not investigate the problem of performing the allreduce operation where the P processes do not form a multi-dimensional rectangular process arrangement. Consider, for example, the execution of SUARA comprising two serial steps. In the first step, SUARA partitions S into R subsets and performs R parallel sub-allreduce operations. In the second step, there will be gaps in the two-dimensional process arrangement if there is no C such that $R \times C = P$. One approach adds processes dynamically and fills the gaps to form a two-dimensional process arrangement, $R \times C = Q, Q > P$. The local value in each new process is set to an appropriate value based on the reduction operator. For example, the local value is set to 0 if the reduction operator is a sum and 1 if the reduction operator is a product. SUARA is then executed using Q processes.

The number of allreduces in SUARA employing P processes and comprising two serial steps ($L = 2$) is $R + C$ where $R \times C = P$. The process partitions are visualized as a two-dimensional process arrangement, $R \times C$. The R horizontal parallel sub-allreduce operations are followed by C vertical parallel sub-allreduce operations or vice versa.

For SUARA consisting of three serial steps ($L = 3$), the number will be $R \times K + C \times K + R \times C$ where $R \times C \times K = P$. Fig. 4 illustrates the three serial steps. The process partitions are visualized as a three-dimensional process arrangement, $R \times C \times K$. In the first step, $R \times K$ parallel sub-allreduce operations are executed horizontally in the C direction. Each sub-allreduce operation involves C processes.

In the second step, $C \times K$ parallel sub-allreduce operations are executed vertically in the R direction. Each sub-allreduce operation involves R processes. In the final step, $R \times C$ parallel sub-allreduce operations take place in the K direction. Each sub-allreduce operation involves K processes. This pattern of communications holds for SUARA for higher dimensions.

In the next section, we design, theoretically study and implement a two-step SUARA called SUARA2 on top of the set of allreduce algorithms in Open MPI.

4. SUARA2: a two-step SUARA on top of the open MPI set of allreduce algorithms

This section describes SUARA2, a two-step SUARA on top of Open MPI library. The processes in SUARA2 form a two-dimensional grid arrangement. The design and implementation of SUARA2 comprise three stages. At the first stage, SUARA2 determines the optimal 2D process grid arrangement and the optimal Open MPI allreduce algorithms to employ in the process rows and columns. At the second stage, process rows execute library allreduce algorithms in parallel. At the third stage, process columns execute library allreduce algorithms in parallel, completing the whole allreduce operation.

We first derive analytical models of six allreduce algorithms used in Open MPI: *linear*, *nonoverlapping tuned reduce followed by*

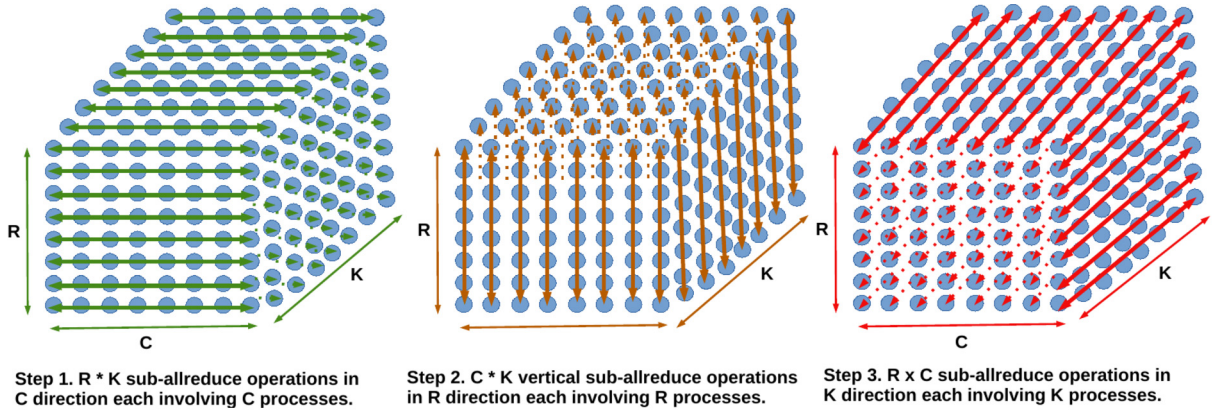


Fig. 4. SUARA consisting of three serial steps. The blue circles represent the processes arranged in a 3D grid arrangement, $R \times C \times K$. In the first step, there are $R \times K$ parallel sub-allreduce operations in the C direction. Each sub-allreduce operation involves C processes and is shown by a green rod joining the processes. In the second step, there are $C \times K$ parallel sub-allreduce operations in the R direction. Each sub-allreduce operation involves R processes and is shown by an orange rod joining the processes. In the final step, there are $R \times C$ parallel sub-allreduce operations in the K direction. Each sub-allreduce operation involves K processes and is shown by a red rod joining the processes.

tuned broadcast, *recursive doubling*, *ring without segmentation*, *ring with segmentation*, and *Rabenseifner*. Then, the derived models are used in SUARA2 for the automatic selection of optimal Open MPI library allreduce algorithms for horizontal and vertical sub-allreduce operations.

4.1. Analytical models of open MPI allreduce algorithms

We present here the six *allreduce* algorithms provided by Open MPI and build their analytical performance models using the basic Hockney model [22] for modelling point-to-point communications.

MPI collective algorithms are commonly implemented using point-to-point communications where the group of processes executing the collective algorithm is mapped into a virtual topology. The virtual topologies include a linear tree, binomial tree, and binary tree, to name a few. MPI libraries use two communication protocols to implement point-to-point communication. They are called *eager* and *rendezvous* used for transferring short and large messages, respectively. We present analytical performance models for the allreduce collective algorithms only for the rendezvous protocol. We assume that each node in the network supports single-port full-duplex communication, which means that a process executing on a node can be involved in a single emission (send) and single reception (receive) simultaneously. The platform employed in this work, Cray XC40 with Aries packet-switched interconnect with Dragonfly topology, satisfies this assumption.

The basic Hockney model is used for modelling a point-to-point communication operation as the fundamental building block of analytical models for allreduce algorithms. The model estimates the time $T_{p2p}(m)$ of sending a message of size m between two processes as $T_{p2p}(m) = \alpha + \beta \cdot m$, where α is the latency, and β is the reciprocal bandwidth. In an allreduce algorithm, the point-to-point communications are followed by computations performing reduction operations on the local vectors. We consider the computation cost per byte of the reduction operation to be γ for any MPI process. The constants (α, β) are considered specific to each allreduce algorithm. The constant γ is independent of an allreduce algorithm but is platform-specific. Each algorithm is executed by P processes in the set, $\{P_0, \cdots, P_{P-1}\}$, with corresponding ranks, $\{0, \cdots, P-1\}$.

To summarize the analytical models that follow, the cost of each algorithm has a latency component given by α, a bandwidth component $(\beta \cdot m)$, and a computation component $(\gamma \cdot m)$. The algorithms that achieve the lower bound for the latency component will have $\log_2 P$ factor. *Recursive doubling* [34] and *Rabenseifner* [35] are two such algorithms. The algorithms that attain the lower bound for the bandwidth component will have $2 \cdot \frac{P-1}{P}$ factor. Rabenseifner falls in this category. Finally, the lower bound for the computation component is $\frac{P-1}{P}$, which is also realized by Rabenseifner. The *linear* and *ring without segmentation* algorithms do not attain the lower bounds for the latency and bandwidth components and, therefore, contain $2 \cdot (P-1)$ factor.

We represent the allreduce algorithms, *linear*, *nonoverlapping tuned reduce followed by tuned broadcast*, *recursive doubling*, *ring without segmentation*, *ring with segmentation*, and *Rabenseifner*, by the short-form identifiers, *linear*, *nono*, *rd*, *rnos*, *rs*, and *rab*, to aid the clarity of our theoretical exposition. The constants (α_a, β_a) correspond to the allreduce algorithm a.

Table 1 contains the analytical formulae for the allreduce algorithms.

4.1.1. Linear

The linear allreduce algorithm is implemented using *linear reduce* algorithm followed by the *linear broadcast* algorithm [34]. Both algorithms transmit a whole message without message segmentation.

Each *receive* in a linear reduce algorithm only starts after the previous one is completed. Therefore, the execution time of the linear reduce algorithm will be equal to the sum of execution times of $(P-1)$ message transmissions. Thus, the execution time of the linear reduce algorithm is as follows:

$$T_{linear\_reduce}(P, m) = (P-1) \cdot (\alpha_{linear} + \beta_{linear} \cdot m + \gamma \cdot m) \quad (1)$$

Each *send* in a linear broadcast algorithm only starts after the previous one is completed. Therefore, the execution time of the linear broadcast algorithm will be equal to the sum of execution times of $(P-1)$ message transmissions. So, the execution time of the linear broadcast algorithm is estimated as follows:

$$T_{linear\_bcast}(P, m) = (P-1) \cdot (\alpha_{linear} + \beta_{linear} \cdot m) \quad (2)$$

Thus, the execution time of the allreduce algorithm is estimated as follows:

$$\begin{aligned} T_{linear}(P, m) &= T_{linear\_reduce}(P, m) + T_{linear\_bcast}(P, m) \\ &= (P-1) \cdot (2 \cdot (\alpha_{linear} + \beta_{linear} \cdot m) + \gamma \cdot m) \end{aligned} \quad (3)$$

4.1.2. Nonoverlapping

The nonoverlapping allreduce algorithm is implemented using the MPI_Reduce operation followed by the MPI_Bcast operation

Table 1
The analytical formulae for the Open MPI allreduce algorithms using the Hockney performance model of communication. The *nonoverlapping* allreduce algorithm has model expressions for thirty different implementations. Therefore, these expressions are not given here due to space constraints.

Allreduce algorithm	Analytical model	Reference
Linear reduce followed by *linear broadcast*	$(P-1)\cdot(2\cdot(\alpha_{linear}+\beta_{linear}\cdot m)+\gamma\cdot m)$	[34]
Recursive doubling	$\log_2 P\cdot(\alpha_{rd}+\beta_{rd}\cdot m+\gamma\cdot m)$	[34]
Ring without segmentation	$2\cdot(P-1)\cdot\alpha_{rnos}+2\cdot\frac{P-1}{P}\cdot\beta_{rnos}\cdot m+\frac{P-1}{P}\cdot\gamma\cdot m$	[34]
Ring with segmentation	$(P+\frac{m}{m_s\times P}-2)\times(\alpha_{rs}+\beta_{rs}\cdot m_s+\gamma\cdot m_s)+(P-1)\cdot(\alpha_{rs}+\beta_{rs}\cdot\frac{m}{P})$	[34]
Rabenseifner	$2\cdot\log_2 P\cdot\alpha_{rab}+2\cdot\frac{P-1}{P}\cdot\beta_{rab}\cdot m+\frac{P-1}{P}\cdot\gamma\cdot m$	[35]

[34]. Both operations are called sequentially. Hence, the allreduce is called nonoverlapping.

$$T_{nono}(P,m)=T_{reduce}(P,m)+T_{bcast}(P,m) \tag{4}$$

The execution times of MPI_Reduce and MPI_Bcast operations are estimated based on the particular collective algorithms employed in their execution. There are six collective algorithms for the MPI_Bcast operation and five for the MPI_Reduce operation in Open MPI. Altogether, it would result in thirty model expressions. We do not present the models here due to space constraints.

4.1.3. Recursive doubling

We will present this allreduce algorithm for the case where P is a power of two [34]. There are $\log_2 P$ steps in the algorithm. In step 1, processes separated by a rank-distance of 1 perform a pairwise exchange of the whole message with each other $(P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, \ldots)$. At the end of the step, both processes in a pairwise exchange redundantly compute the same partial reduction of the whole message. In step 2, the distance is doubled, and the pairwise exchanges are $(P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, \ldots)$. In the last step $\log_2 P$, the processes will be separated by distance $\frac{P}{2}$. At the end of the final communication step, the processes compute the final partial reduction of the whole message, thereby completing the whole allreduce operation.

Therefore, given the assumption of full-duplex communication, the algorithm's execution time is estimated as follows:

$$T_{rd}(P,m)=\log_2 P\cdot(\alpha_{rd}+\beta_{rd}\cdot m+\gamma\cdot m) \tag{5}$$

4.1.4. Ring without segmentation

This allreduce algorithm is implemented using a reduce_scatter operation followed by an allgather operation [34]. Both operations are performed using a logical ring communication pattern, $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \cdots \rightarrow P_{P-1} \rightarrow P_0$. Each process has a left neighbour and a right neighbour. For example, process P_0 has processes P_{P-1} and P_1 as its left and right neighbours, respectively. There are $P-1$ ring exchange steps in each operation. The message of size m at each process is split into P chunks, $\{S_0, \cdots, S_{P-1}\}$, each of size $\frac{m}{P}$. The algorithm is described in detail in the supplemental, Section 4. The algorithm's execution time is estimated as follows:

$$\begin{aligned}T_{rnos}(P,m)&=2\cdot(P-1)\cdot\alpha_{rnos}+2\cdot\frac{P-1}{P}\cdot\beta_{rnos}\cdot m\\&\quad+\frac{P-1}{P}\cdot\gamma\cdot m\end{aligned} \tag{6}$$

4.1.5. Ring with segmentation

Like the *ring without segmentation* algorithm, this allreduce algorithm is implemented using a reduce_scatter operation followed by an allgather operation [34]. Both operations are performed using a logical ring communication pattern. First, the message of size m at each process is split into P chunks each of size $\frac{m}{P}$. Each chunk is further broken into n_s number of segments of size, $m_s=\frac{m}{n_s\times P}$. Therefore, instead of the chunks, the segments are communicated by the processes in the reduce_scatter and allgather operations.

The execution time of the algorithm is estimated as follows:

$$\begin{aligned}T_{rs}(P,m,m_s)&=(P+\frac{m}{m_s\times P}-2)\times(\alpha_{rs}+\beta_{rs}\cdot m_s+\gamma\cdot m_s)\\&\quad+(P-1)\cdot(\alpha_{rs}+\beta_{rs}\cdot\frac{m}{P})\end{aligned} \tag{7}$$

The *Ring without segmentation* allreduce algorithm is a special case of this algorithm where $n_s=1, m_s=\frac{m}{P}$.

4.1.6. Rabenseifner

The Rabenseifner algorithm involves a reduce_scatter operation followed by an allgather operation [35]. The reduce_scatter operation is implemented using a recursive data halving and rank-distance doubling algorithm. The allgather operation is implemented using recursive data doubling and rank-distance halving algorithm. The cost for the case where P is a power of two is detailed in the supplemental, Section 4. The algorithm's execution time is estimated as follows:

$$\begin{aligned}T_{rab}(P,m)&=2\cdot\log_2 P\cdot\alpha_{rab}+2\cdot\frac{P-1}{P}\cdot\beta_{rab}\cdot m\\&\quad+\frac{P-1}{P}\cdot\gamma\cdot m\end{aligned} \tag{8}$$

4.2. SUARA2: description of the algorithm

In this section, we present SUARA2, a two-step SUARA implemented on the top of the Open MPI set of allreduce algorithms. The inputs to SUARA2 include the six standard arguments to the MPI_Allreduce function, the starting address of send buffer, *sendbuf*; number of elements in send buffer, *count*; the MPI data type of elements of send buffer, *datatype*; the MPI operation, *op*; and the MPI communicator, *comm*. The other input parameters are the message segment size, m_s; the set of Open MPI allreduce algorithms, A; the time of computation per byte, γ; and a (α, β) pair for each Open MPI allreduce algorithm (described in the previous section). Each process contains a local vector of values (size equal to m bytes) in the send buffer, *sendbuf*, input to SUARA2 for reduction. The number of available processes, P, is the size of the input MPI communicator, *comm*.

SUARA2 comprises three stages. In the first stage, SUARA2 determines the optimal 2D process grid arrangement of the set of P processes, (P_r, P_c), $P=P_r\times P_c$, and the optimal Open MPI allreduce algorithms to employ in the process rows and columns. The second stage executes the horizontal sub-allreduce operations in parallel in the P_r process rows. Finally, the third stage executes the vertical sub-allreduce operations in parallel in the P_c process columns, thereby completing the allreduce operation.

Fig. 5 illustrates the execution of SUARA2 for an example where $P=9$ and A signifying the set of Open MPI allreduce algorithms. In the first stage, SUARA2 determines the optimal 2D process grid

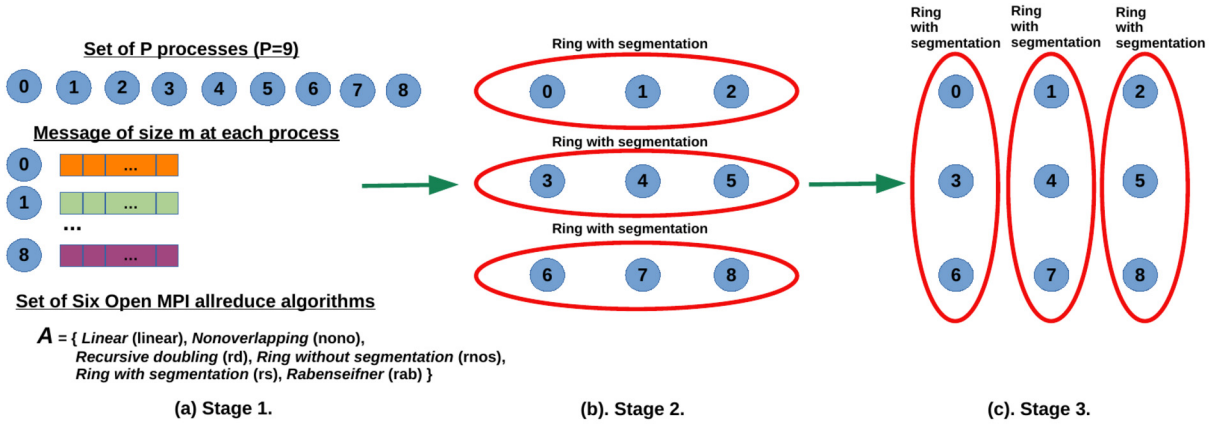


Fig. 5. Execution of SUARA2 for the inputs $P = 9$, A, and m. Blue circles signify processes with ranks inside the circle. In the first stage, it determines the optimal 2D process grid arrangement $(3, 3)$ and the optimal allreduce algorithms to employ in the process rows and columns. The second and third stages of SUARA2 involve execution of *Ring with segmentation* algorithms in the process rows and columns. The red rings represent the execution of *Ring with segmentation* algorithms. The sub-allreduce operations in the process rows occur in parallel followed by parallel sub-allreduce operations in the process columns.

arrangement ($(P_r, P_c) = (3, 3)$) and the optimal allreduce algorithms in the process rows and columns (*Ring with segmentation*, *Ring with segmentation*). The second and third stages involve the execution of *Ring with segmentation* allreduce algorithms in process rows and columns. The whole allreduce operation is completed after the execution of all three stages.

Let us consider the first stage of the SUARA2 execution. One approach to determine the optimal 2D process grid arrangement is an exhaustive search that estimates the execution times of all the possible 2D process grid arrangements, (P_r, P_c), $P = P_r \times P_c$, for a given set of P processes. For each 2D process grid arrangement, (P_r, P_c), the sums of the execution times are estimated for all possible combinations of allreduce algorithms in the process rows and all possible combinations of allreduce algorithms in the process columns. Then, the allreduce algorithmic combinations in the process rows and columns that yield the minimum sum are selected for this process grid arrangement. Finally, the 2D process grid arrangement that results in a minimum estimated execution time is output from this stage. However, the exhaustive approach is infeasible due to the exponential number of allreduce algorithmic combinations.

Fortunately, we do not have to consider all the possible combinations. Indeed, suppose algorithm a is estimated to be the fastest allreduce algorithm for one row of P_c processes. In that case, it will also be fastest for all other rows as the estimated time of any allreduce algorithm given by formulae (1)–(8) only depends on the number of processes in the row, the message size, and possibly the segment size, and the values of these parameters are the same for all rows. Therefore, using algorithm a in all rows will give us the fastest parallel execution of row-wise allreduce sub-operations. Any other combination will be slower as the time of parallel execution equals the time of the slowest algorithm in the combination. Similarly, using the same, fastest, algorithm in all columns will give us the fastest parallel execution of column-wise allreduce sub-operations.

Thus, in its first stage, SUARA2 must only examine allreduce combinations given by a pair of identifiers, (a_r, a_c), representing the allreduce algorithms employed in the process rows and columns, respectively. The identifiers, a_r and a_c, take values in the set, {*linear*, *nono*, *rd*, *rnos*, *rs*, *rab*}, which are the short forms for the Open MPI allreduce algorithms. Hence, there will be only 36 combinations to examine, six with the same algorithm in process rows and columns and thirty with different algorithms.

Next, for each of the thirty-six allreduce combinations we derive a cost analytical model, which allows us to calculate the 2D grid arrangement of processes, (P_r, P_c), optimal for this combination, as well as the execution time of the combined allreduce operation using this arrangement and algorithmic combination.

We proceed as follows. For given P, P_c, m, m_s, a_r, and a_c, the execution time of the combined allreduce operation can be expressed as follows:

$$T_{SUARA2}(P, P_c, m, m_s, a_r, a_c) = T_{a_r}(P_c, m, m_s) + T_{a_c}(\frac{P}{P_c}, m, m_s) \tag{9}$$

Here, T_{a_r} gives the cost of the allreduce operations in the process rows using the allreduce algorithm given by the identifier, a_r. T_{a_c} represents the cost of the allreduce operations in the process columns employing the allreduce algorithm given by the identifier, a_c.

For given P, m, m_s, a_r, and a_c, T_{SUARA2} is a discrete function of P_c with the domain, $P_c \in \{1, 2, \cdots, P\}$. We analyze the extension of this function in the real domain, $P_c \in [1, P]$, represented by the same analytical expression for T_{SUARA2}.

By its definition, the extended T_{SUARA2} has the following properties:

- Since the input parameters, P, m, m_s, γ and the (α, β) pairs, are positive, T_{SUARA2} is a positive function of P_c, $T_{SUARA2} : [1, P] \rightarrow \mathbb{R}_{>0}$.
- T_{SUARA2} is a sum of logarithmic, linear, and reciprocal functions of P_c. Therefore, it is continuous and has continuous first and second derivatives in the interval, $[1, P]$.

Consider the values of T_{SUARA2} at the endpoints, $\{1, P\}$. $T_{SUARA2}(P, 1, m, m_s, a_r, a_c)$ gives the estimated execution time of the a_c allreduce algorithm employing a linear arrangement of P processes. $T_{SUARA2}(P, P, m, m_s, a_r, a_c)$ gives the estimated execution time of the a_r allreduce algorithm employing a linear arrangement of P processes. Therefore, if $a_r = a_c$, then $T_{SUARA2}(P, 1, m, m_s, a_r, a_c)$ and $T_{SUARA2}(P, P, m, m_s, a_r, a_c)$ are equal.

If message segmentation is not employed, then m is equal to m_s and $T_{SUARA2}(P, P_c, m, m_s, a_r, a_c)$ is equal to $T_{SUARA2}(P, P_c, m, m, a_r, a_c)$.

To derive the formula calculating the optimal (P_r, P_c) for a given (a_r, a_c), we analyze T_{SUARA2} using the traditional calculus approach to determine the optimal value of P_c that minimizes the function. The main steps of this analysis follow:

- If T_{SUARA2} is a constant function of P_c, then the estimated execution time is the same for all process arrangements. There-

fore, we select one of the process arrangements, $(P_r, P_c) = (1, P)$. The analytical formula returns $(1, P)$, and the estimated execution time, $T_{SUARA2}(P, P, m, m_s, a_r, a_c)$.

- To determine the stationary point (P_c^*) of T_{SUARA2} in the interval $[1, P]$, we obtain its first and second derivatives as follows:

$$\begin{aligned}\frac{\partial T_{SUARA2}}{\partial P_c} &= \frac{\partial T_{a_r}(P_c, m, m_s)}{\partial P_c} + \frac{\partial T_{a_c}(\frac{P}{P_c}, m, m_s)}{\partial P_c} \\ \frac{\partial^2 T_{SUARA2}}{\partial P_c^2} &= \frac{\partial^2 T_{a_r}(P_c, m, m_s)}{\partial P_c^2} + \frac{\partial^2 T_{a_c}(\frac{P}{P_c}, m, m_s)}{\partial P_c^2}\end{aligned} \tag{10}$$

- We solve the equation $\frac{\partial T_{SUARA2}}{\partial P_c} = 0$ to determine the stationary point (P_c^*).
- We found that the sign of the second derivative $\frac{\partial^2 T_{SUARA2}}{\partial P_c^2}$ does not depend on P_c but only depends on the input parameters, $(P, m, m_s, \gamma, \alpha_{a_r}, \beta_{a_r}, \alpha_{a_c}, \beta_{a_c})$. We illustrate this for the various allreduce combinations that we analyze below. Therefore, $\frac{\partial^2 T_{SUARA2}}{\partial P_c^2}$ is either positive or negative or zero in the interval $[1, P]$.
- If $\frac{\partial^2 T_{SUARA2}}{\partial P_c^2} > 0$, then P_c^* minimizes T_{SUARA2}. We then consider two integer approximations of P_c^*, $\lfloor P_c^* \rfloor$ and $\lceil P_c^* \rceil$ (the floor and ceiling). If $T_{SUARA2}(P, \lfloor P_c^* \rfloor, m, m_s, a_r, a_c) < T_{SUARA2}(P, \lceil P_c^* \rceil, m, m_s, a_r, a_c)$, then $(P_r, P_c) = (\frac{P}{\lfloor P_c^* \rfloor}, \lfloor P_c^* \rfloor)$ and $T_{SUARA2}(P, \lfloor P_c^* \rfloor, m, m_s, a_r, a_c)$. Otherwise, $(P_r, P_c) = (\frac{P}{\lceil P_c^* \rceil}, \lceil P_c^* \rceil)$ and $T_{SUARA2}(P, \lceil P_c^* \rceil, m, m_s, a_r, a_c)$.
- If $\frac{\partial^2 T_{SUARA2}}{\partial P_c^2} \leq 0$, the value of P_c that minimizes T_{SUARA2} is one of or both the endpoints, $\{1, P\}$. Therefore, the optimal process grid arrangement is a linear arrangement of P processes.
  - If the allreduce algorithms employed in the process rows (a_r) and columns (a_c) are the same $(a_r = a_c)$, then either of the endpoints minimize T_{SUARA2}. Therefore, we select one of the process arrangements, $(P_r, P_c) = (1, P)$. The analytical formula returns $(1, P)$, and the estimated execution time, $T_{SUARA2}(P, P, m, m_s, a_r, a_c)$.
  - If the allreduce algorithms employed in the process rows (a_r) and columns (a_c) are different $(a_r \neq a_c)$, then the estimated execution times for the two endpoints are considered. If $T_{SUARA2}(P, 1, m, m_s, a_r, a_c) < T_{SUARA2}(P, P, m, m_s, a_r, a_c)$, then the analytical formula returns $(P, 1)$ and $T_{SUARA2}(P, 1, m, m_s, a_r, a_c)$. Otherwise, it returns $(1, P)$ and $T_{SUARA2}(P, P, m, m_s, a_r, a_c)$.

We demonstrate how the method works using one particular allreduce combination, {*Ring with segmentation*, *Ring with segmentation*}. We chose this combination for illustration because it appeared optimal in our experimental setup. In the supplemental file, we present derivations for six other allreduce combinations.

4.2.1. Ring with segmentation, ring with segmentation

$$\begin{aligned}&T_{SUARA2}(P, P_c, m, m_s, rs, rs) = T_{rs}(P_c, m, m_s) + T_{rs}(\frac{P}{P_c}, m, m_s) \\ &= (P_c + \frac{m}{m_s \cdot P} - 2) \cdot (\alpha_{rs} + \beta_{rs} \cdot m_s + \gamma \cdot m_s) \\ &+ (P_c - 1) \cdot (\alpha_{rs} + \beta_{rs} \cdot \frac{m}{P_c}) \\ &+ (\frac{P}{P_c} + \frac{m}{m_s \cdot P} - 2) \cdot (\alpha_{rs} + \beta_{rs} \cdot m_s + \gamma \cdot m_s) \\ &+ (P/P_c - 1) \cdot (\alpha_{rs} + \beta_{rs} \cdot \frac{m \cdot P_c}{P})\end{aligned}$$

$$\begin{aligned}&\frac{\partial T_{SUARA2}}{\partial P_c} = (1 - \frac{P}{P_c^2}) \cdot (2 \cdot \alpha_{rs} + \beta_{rs} \cdot m_s + \gamma \cdot m_s - \frac{\beta_{rs} \cdot m}{P}) \\ &\frac{\partial T_{SUARA2}}{\partial P_c} = 0 \quad \Longrightarrow \quad P_c = \sqrt{P} \\ &\text{if } (A_1 = (2 \cdot \alpha_{rs} + \beta_{rs} \cdot m_s + \gamma \cdot m_s - \frac{\beta_{rs} \cdot m}{P}) \neq 0) \\ &\frac{\partial^2 T_{SUARA2}}{\partial P_c^2} = 2 \cdot \frac{A_1}{P_c^3}\end{aligned} \tag{11}$$

Therefore, if $A_1 > 0$, $P_c = \sqrt{P}$ minimizes T_{SUARA2}. The integer approximation ($\lfloor \sqrt{P} \rfloor$ or $\lceil \sqrt{P} \rceil$) that gives the least estimated execution time is then chosen. Therefore, if $T_{SUARA2}(P, \lfloor \sqrt{P} \rfloor, m, m, rs, rs) < T_{SUARA2}(P, \lceil \sqrt{P} \rceil, m, m, rs, rs)$, the chosen process arrangement will be $(P_r, P_c) = (\frac{P}{\lfloor \sqrt{P} \rfloor}, \lfloor \sqrt{P} \rfloor)$, and the corresponding estimated execution time will be $T_{SUARA2}(P, \lfloor \sqrt{P} \rfloor, m, m, rs, rs)$. Otherwise, it will be $(P_r, P_c) = (\frac{P}{\lceil \sqrt{P} \rceil}, \lceil \sqrt{P} \rceil)$ and $T_{SUARA2}(P, \lceil \sqrt{P} \rceil, m, m, rs, rs)$.

If $A_1 \leq 0$, the value of P_c that minimizes T_{SUARA2} is either of the endpoints, $\{1, P\}$. Therefore, the chosen process arrangement and the corresponding estimated execution time will be $(P_r, P_c) = (1, P)$ and $T_{SUARA2}(P, P, m, m, rs, rs)$ respectively. Thus, the derived analytical formula will be as follows:

$$\begin{cases} \textbf{if } A_1 > 0 \\ \quad \textbf{if } T_{SUARA2}(P, \lfloor \sqrt{P} \rfloor, m, m, rs, rs) \\ \quad\quad < T_{SUARA2}(P, \lceil \sqrt{P} \rceil, m, m, rs, rs) \\ \quad\quad\quad (P_r, P_c) = (\frac{P}{\lfloor \sqrt{P} \rfloor}, \lfloor \sqrt{P} \rfloor), T_{SUARA2}(P, \lfloor \sqrt{P} \rfloor, m, m, rs, rs) \\ \quad \textbf{else } (P_r, P_c) = (\frac{P}{\lceil \sqrt{P} \rceil}, \lceil \sqrt{P} \rceil), T_{SUARA2}(P, \lceil \sqrt{P} \rceil, m, m, rs, rs) \\ \textbf{else } (P_r, P_c) = (1, P), T_{SUARA2}(P, P, m, m, rs, rs) \end{cases} \tag{12}$$

In total, thirty-six analytical formulae for all algorithmic combinations are derived this way. These formulae are then used at runtime for efficient selection of the optimal algorithmic combination and the optimal 2D process arrangement corresponding to the optimal algorithmic combination.

4.3. SUARA2: pseudocode

Algorithm 1 illustrates the execution of the three main stages of SUARA2. The first six parameters are the standard parameters of the MPI_Allreduce function. The input parameter, $H_{\alpha\beta}$, represents the set of Hockney model (α, β) pairs for each Open MPI allreduce algorithm in the set of allreduce algorithms, A. The output parameters are the starting address of receive buffer, $recvbuf$; the optimal 2D process grid arrangement, (P_r, P_c); the optimal allreduce combination, (a_r, a_c), the optimal estimated executed time of SUARA2, T_{SUARA2}, and the MPI return code, *status*.

Lines 3–6 contain the main stage 1 steps of SUARA2. First, the number of available processes, P, is obtained using the MPI function, MPI_Comm_size, using the input MPI communicator, *comm*. Next, the message size m is calculated using the user function, $GetMessageSize$, based on the inputs, *count* and *datatype*.

The user function, $SUARA2\_find\_optimals$, determines the optimal allreduce combination, (a_r, a_c), the optimal 2D process arrangement, (P_r, P_c), and the optimal estimated execution time of SUARA2, T_{SUARA2}, given the inputs, $P, m, m_s, A, H_{\alpha\beta}$ and γ.

The Algorithm 2 depicts the main steps of $SUARA2\_find\_optimals$. For each (a_r, a_c) combination, the optimal 2D process grid arrangement (P_r, P_c) and the optimal estimated SUARA2 execution time, T, are obtained using the analytical formula, $AFormula$. The manual derivation of the analytical formula for

Algorithm 1 Pseudocode illustrating the three main stages of SUARA2.

1: **procedure** SUARA2($sendbuf, recvbuf, count, datatype, op, comm, m_s, A, H_{\alpha\beta}, \gamma, P_r, P_c, a_r, a_c, T_{SUARA2}, status$)
Input:
Starting address of send buffer, $sendbuf$
Number of elements in send buffer, $count$
MPI Data type of elements of send buffer, $datatype$
MPI Operation, op
MPI communicator, $comm$
Message segment size, m_s
Set of Open MPI allreduce algorithms, $A = \{linear, \cdots, rab\}$
Set of (α, β) pairs for the allreduce algorithms, $H_{\alpha\beta} = \{(\alpha_{linear}, \beta_{linear}), \cdots, (\alpha_{rab}, \beta_{rab})\}$
Time of computation per byte, γ
Output:
Starting address of receive buffer, $recvbuf$
Optimal 2D process grid arrangement, (P_{ropt}, P_{copt})
Optimal allreduce combination, (a_{ropt}, a_{copt})
Optimal estimated executed time of SUARA2, T_{SUARA2}
Return code, $status$

2: /* **First stage of SUARA2** */
3: $Iam \leftarrow MPI\_Comm\_rank(comm)$
4: $P \leftarrow MPI\_Comm\_size(comm)$
5: $m \leftarrow GetMessageSize(count, datatype)$
6: $(P_r, P_c, a_r, a_c, T_{SUARA2}) \leftarrow SUARA2\_find\_optimals(P, m, m_s, A, H_{\alpha\beta}, \gamma)$
7: /* **Second stage of SUARA2** */
8: $(myr, myc) \leftarrow GetRowColRanks(Iam, P, P_r, P_c)$
9: $SetAllreduceAlgorithmParams(a_r, m_s)$
10: $status \leftarrow MPI\_Comm\_split(comm, myr, myc, rowcomm)$
11: $status \leftarrow MPI\_Allreduce(sendbuf, count, datatype, op, rowcomm)$
12: $status \leftarrow MPI\_Comm\_free(rowcomm)$
13: /* **Final stage of SUARA2** */
14: $SetMPIAllreduceRuntimeParams(a_c, m_s)$
15: $status \leftarrow MPI\_Comm\_split(comm, myc, myr, colcomm)$
16: $status \leftarrow MPI\_Allreduce(sendbuf, count, datatype, op, colcomm)$
17: $status \leftarrow MPI\_Comm\_free(colcomm)$
18: **return** $(P_r, P_c, a_r, a_c, T_{SUARA2}, status)$
19: **end procedure**

Algorithm 2 Algorithm to determine the optimal allreduce combination, 2D process arrangement, and the estimated execution time of SUARA2.

1: **procedure** SUARA2_FIND_OPTIMALS($P, m, m_s, A, H_{\alpha\beta}, \gamma, P_{ropt}, P_{copt}, a_{ropt}, a_{copt}, T_{SUARA2}$)
Input:
Number of processes, P
Message size, m
Message segment size, m_s
Set of Open MPI allreduce algorithms, $A = \{linear, \cdots, rab\}$
Set of (α, β) pairs for the allreduce algorithms, $H_{\alpha\beta} = \{(\alpha_{linear}, \beta_{linear}), \cdots, (\alpha_{rab}, \beta_{rab})\}$
Time of computation per byte, γ
Output:
Optimal 2D process grid arrangement, (P_{ropt}, P_{copt})
Optimal allreduce combination, (a_{ropt}, a_{copt})
Optimal estimated executed time of SUARA2, T_{SUARA2}

2: $T_{SUARA2} \leftarrow \infty$
3: **for** $a_r \in A$ **do**
4: **for** $a_c \in A$ **do**
5: $(P_r, P_c, T) \leftarrow AFormula(P, m, m_s, a_r, a_c, H_{\alpha\beta}, \gamma)$
6: **if** $(T < T_{SUARA2})$ **then**
7: $P_{ropt} \leftarrow P_r$; $P_{copt} \leftarrow P_c$
8: $a_{ropt} \leftarrow a_r$; $a_{copt} \leftarrow a_c$
9: $T_{SUARA2} \leftarrow T$
10: **end if**
11: **end for**
12: **end for**
13: **return** $(P_{ropt}, P_{copt}, a_{ropt}, a_{copt}, T_{SUARA2})$
14: **end procedure**

an allreduce combination is described in detail in the previous section. Thus, the function, $SUARA2\_find\_optimals$, determines and returns the best allreduce combination, (a_{ropt}, a_{copt}), and the optimal 2D process arrangement, (P_{ropt}, P_{copt}), corresponding to (a_{ropt}, a_{copt}) that results in minimal estimated SUARA2 execution time, T_{SUARA2}.

Lines 8–12 present the second stage steps of SUARA2. In Line 8, each process obtains its coordinates, (myr, myc), in the 2D process grid arrangement, (P_r, P_c), using the user function, $GetRowColRanks$. Line 9 invokes the user function, $SetMPIAllreduceRuntimeParams$, that calls the MPI library implementation-specific runtime functions to set the allreduce algorithm a_r to be employed during the execution of $MPI\_Allreduce$ and message segment size to use if the allreduce algorithm is *Ring with segmentation*. The row communicators employed for the horizontal sub-allreduce operations are obtained using the MPI library function, MPI_Comm_split, in Line 10. There will be P_r row subcommunicators created in Line 10 because P_r different colours, myr, are passed in the second argument to MPI_Comm_split. At Line 11, P_r horizontal sub-allreduce operations employing the a_r allreduce algorithm happen in parallel in the process rows using the P_r row subcommunicators.

Lines 14–17 show the final stage steps of SUARA2, completing the whole allreduce operation. First, the user function, $SetMPIAllreduceRuntimeParams$, sets the allreduce algorithm a_c and message segment size to use during the execution of $MPI\_Allreduce$ (Line 14). Next, the column communicators employed for the vertical sub-allreduce operations are created using the MPI library function, MPI_Comm_split, in Line 15. There will be P_c column subcommunicators created in Line 17 since P_c different colours, myc, are passed in the second argument to MPI_Comm_split. Finally, at Line 18, P_c vertical sub-allreduce operations employing the a_c allreduce algorithm occur in parallel in the process columns using the P_c column subcommunicators thereby completing the allreduce operation.

4.4. Accuracy of estimation using theoretical models

Castelló et al. [8] highlight the factors that negatively affect the accuracy of estimation of execution times of Open MPI allreduce algorithms using theoretical models.

We make sure we minimize the negative impact on the accuracy of selection of the optimal allreduce combination in SUARA2 using theoretical models by following the steps below:

- Deriving analytical models for the different Open MPI allreduce algorithms from the Open MPI code implementing the algorithms rather than from high-level mathematical definitions. The analytical models consider the algorithms' properties, which significantly impact their performance and can only be extracted from the implementation code. Such properties include blocking or non-blocking, rendezvous or eager protocol, and segmentation/pipelining.
- Employing a different pair (α_a, β_a) for a collective algorithm, a, and accurately estimating each pair using careful design of the communication experiments. Nuriyev et al. [30,31] found that using different (α_a, β_a) for a collective algorithm, a, improves the accuracy of selection of the best performing collective algorithm from the set of native MPI collective algorithms for a collective operation. The insight behind using algorithmic-specific (α_a, β_a) pairs is that the estimated values of the α_a and β_a capture, not just network characteristics but also algorithm-specific traits. More specifically, a specific communication experiment is designed for each collective algorithm so that the algorithm itself would be involved in the execution of the experiment. Moreover, the execution time of this experiment must be dominated by the execution time of this collective algorithm.
- Determining the optimal 2D process arrangement using the traditional calculus approach for each allreduce combination

employing different native allreduce algorithms in process rows and columns. SUARA2 then selects the optimal allreduce combination out of the thirty-six allreduce combinations using the optimal 2D process arrangement determined for each allreduce combination and employs this optimal combination for the execution of the MPI allreduce collective operation.

- Nuriyev et al. [31] demonstrate that the basic Hockney model is inaccurate and needs to be revised for one-process-per-core application configuration on multicore clusters since it does not consider network congestion (contention). Therefore, we focus on only one-process-per-node application configuration in this work. Based on experiments on our experimental platform (Shaheen-II Cray CX40), we observed that the basic Hockney model, by following the above steps, is accurate for this application configuration.

4.5. SUARA2: runtime efficiency, implementation specifics, and portability

The runtime efficiency of SUARA2 is determined by the efficiency of the $SUARA2\_find\_optimals$ function (Algorithm 1, Line 6), which selects the optimal algorithmic combination, the creation of the row and column subcommunicators (Lines 10 and 15), and the freeing of row and column subcommunicators (Lines 12 and 17). The $SUARA2\_find\_optimals$ function is very efficient since it uses only thirty-six analytical formulae to select the optimal algorithmic combination where each formula evaluates a simple condition and an analytical expression. Furthermore, we found that the creation of the row and column subcommunicators using the MPI library function, MPI_Comm_split, and the freeing of the subcommunicators using MPI_Comm_free are not expensive on our experimental platform, a Cray XC40 supercomputer, even for large P. The execution times of $SUARA2\_find\_optimals$ observed in our experiments are in microseconds compared to the execution times of SUARA2, which range from milliseconds to a few seconds for large P.

The message segment size, m_s, input to SUARA2 is typically obtained from the set of recommended values from practice for Open MPI. For example, the default value of the MCA parameter, *btl_sm_max_send_size*, is known to give good performance. The set of Open MPI algorithms, A, is determined by querying the Open MPI Modular Component Architecture (MCA) tuning interface functions. The Hockney model pairs, $\{(\alpha_{linear}, \beta_{linear}), \cdots, (\alpha_{rab}, \beta_{rab})\}$, for the allreduce algorithms in A and the time of computation per byte, γ, are determined offline and input to SUARA2.

The function, $SetMPIAllreduceRuntimeParams$, employs the MPI_T interface functions to set the allreduce algorithm and segment size at the runtime. MPI_T interface functions are introduced in MPI 3.0 that allow getting and setting performance and control variables exposed by an MPI implementation. The implementation of $SetMPIAllreduceRuntimeParams$ is provided in the supplemental.

This section described the design and implementation of SUARA2 on top of the Open MPI set of allreduce algorithms. However, this process is highly portable to other open-source MPI implementations that provide a set of allreduce algorithms whose analytical models are either published or can be derived from their sources.

4.6. Theoretical speedup of SUARA2 over the best open MPI native allreduce algorithm

We analyse the theoretical speedup of SUARA2 over the best Open MPI native allreduce algorithm.

The best Open MPI native allreduce algorithm is specific to a platform and depends on many parameters, that include the number of processes executing the algorithm, P, the message size, m, the message segment size, m_s, the time of computation per byte, γ, and the (α, β) pair that is specific to an allreduce algorithm. Based on our experiments, we find that this algorithm on our experimental platform, a Cray XC40 supercomputer, is *Ring with segmentation* algorithm for $P > 64$. We also observe that SUARA2 employs the allreduce combination, (rs, rs), that uses *ring with segmentation* algorithm in the process rows and columns for $P > 64$. SUARA2 uses the 2D process arrangement $\frac{P}{\sqrt{P}} \times \sqrt{P}$, which is optimal for this allreduce combination.

Therefore, we illustrate the speedup for the case where the best Open MPI native allreduce algorithm is *Ring with segmentation* algorithm, and SUARA2 employs *Ring with segmentation* algorithm in the process rows and columns and $A_1 > 0$. The speedup is equal to the ratio of the cost of the Open MPI algorithm executed by P processes and the cost of SUARA2 employing the allreduce combination (rs, rs) and 2D process grid arrangement, $\frac{P}{\sqrt{P}} \times \sqrt{P}$.

The cost of SUARA2 is

$$\begin{aligned} &T_{SUARA2}(P, \sqrt{P}, m, m_s, rs, rs) \\ &= 2 \cdot (\alpha_{rs} + \beta_{rs} \cdot m_s + \gamma \cdot m_s) \cdot (\sqrt{P} + \frac{m}{m_s \cdot \sqrt{P}} - 2) \\ &+ 2 \cdot (\sqrt{P} - 1) \cdot (\alpha_{rs} + \frac{\beta_{rs} \cdot m}{\sqrt{P}}) \end{aligned} \tag{13}$$

The cost of the Open MPI algorithm is,

$$\begin{aligned} &T_{OpenMPI}(P, m, m_s, rs) \\ &= (\alpha_{rs} + \beta_{rs} \cdot m_s + \gamma \cdot m_s) \cdot (P + \frac{m}{m_s \cdot P} - 2) \\ &+ (P - 1) \cdot (\alpha_{rs} + \frac{\beta_{rs} \cdot m}{P}) \end{aligned} \tag{14}$$

Expressing asymptotically for large P, the costs become

$$\begin{aligned} T_{SUARA2}(P, \sqrt{P}, m, m_s, rs, rs) &= 2(A_2\sqrt{P} - \frac{A_3}{\sqrt{P}} + A_4) \\ &= \mathcal{O}(\sqrt{P}) \\ T_{OpenMPI}(P, m, m_s, rs) &= A_2 P - \frac{A_3}{P} - A_4 \\ &= \mathcal{O}(P) \end{aligned} \tag{15}$$

where A_2 and A_3 are expressions without P,

$$\begin{aligned} A_2 &= 2 \cdot \alpha_{rs} + \beta_{rs} \cdot m_s + \gamma \cdot m_s \\ A_3 &= (\alpha_{rs} + \gamma \cdot m_s) \cdot \frac{m}{m_s} \\ A_4 &= (\alpha_{rs} + \beta_{rs} \cdot m_s + \gamma \cdot m_s) \cdot 2 + \alpha_{rs} - \beta_{rs} \cdot m \end{aligned} \tag{16}$$

The asymptotic theoretical speedup of SUARA2 is,

$$\begin{aligned} \frac{T_{OpenMPI}(P, m, m_s, rs)}{T_{SUARA2}(P, \sqrt{P}, m, m_s, rs, rs)} &= \frac{\mathcal{O}(P)}{\mathcal{O}(\sqrt{P})} \\ &= \mathcal{O}(\sqrt{P}) \end{aligned} \tag{17}$$

5. Experimental results

We demonstrate the practical efficiency of SUARA2 in this section. Our experimental platform is Shaheen II, a Cray XC40 supercomputer. The supercomputer has 6,174 compute nodes, and each node comprises a dual-socket 16-core Intel Haswell processor running at 2.3 GHz. In addition, each node has 128 GB of DDR4 memory running at 2300 MHz. Only 1024 nodes are used for our experiments. The supercomputer does not contain graphics processing units (GPUs). Therefore, only CPUs are used in our

Table 2
Specification of Cray XC40 Shaheen-II.

Specification	Description
Node	Processor type: Intel Haswell, 2 CPU sockets per node, 16 processor cores per CPU, 2.3 GHz
No. of nodes	6174
Memory	128 GB per node, Over 790 TB total memory
Network	Cray Aries interconnect with Dragonfly topology
Storage	Sonexion 2000 Lustre, 17.6 PB of usable storage, Over 500 GB/s bandwidth

Table 3
The settings used for ResNet-50 training.

Parameter	Value
Training batch size	32
Validation batch size	32
No. of Epochs	90
SGD momentum	0.9
Weight decay	0.00005

Table 4
The values of (α, β) experimentally obtained on Shaheen-II for the allreduce algorithm, *ring with segmentation*. The parameter, γ, is the time of summation reduction computation per byte.

Parameter	Value
α_{rs}	1.5e-06
β_{rs}	6.25e-11
γ	2e-10

experiments. The compute nodes are connected via the Cray Aries High Speed Network. Aries is a packet-switched interconnect with a dragonfly topology comprising 18 groups of nodes where nodes within a group are interconnected with a 2D all-to-all structure. The data storage is a Lustre Parallel file system based on Cray Sonexion 2000 with a storage capacity of 17.2 PB delivering around 500 GB/s of I/O throughput. The specification of the supercomputer is shown in Table 2.

We got access to use Shaheen-II only for a short duration, which limited the extent and scope of our experiments.

Horovod with Open MPI is used for training ResNet-50 DNN using ImageNet ILSVRC2010 dataset [37]. The Horovod and Open MPI versions are 0.19.2 and 4.0.3. The steps to install them on Shaheen-II are given in the supplemental. Open MPI is configured with "–with-slurm" option to interface with Slurm job scheduler.

To determine the execution times of SUARA2, Horovod is compiled by replacing the invocation to the native Open MPI allreduce routine with our Open MPI wrapper containing the SUARA2 implementation. The Slurm Open MPI script deployed to execute the Horovod ResNet-50 MPI application is given in the supplemental.

The MPI application is multi-threaded that executes one process per node employing a number of threads equal to the number of cores in a node. While the basic Hockney model employed in the design and implementation of SUARA2 is accurate for one-process-per-node application configuration, it is inaccurate for one-process-per-core application configuration on multicore clusters since it does not consider network congestion (contention). Hence, the basic Hockney model must be either revised or a more accurate analytical model must be employed [31]. *Therefore, application configurations that include one-process-per-socket and one-process-per-core are out of the scope of this work.*

The settings employed for ResNet-50 training [17] are given in Table 3. The ResNet-50 is a deep convolutional network containing 50 layers described in detail in the supplemental. The ImageNet dataset is a large collection of images organized according to the WordNet hierarchy. Each meaningful concept in WordNet, possibly described by multiple words or word phrases, is called a "synonym set" or "synset." There are more than 100,000 synsets in WordNet, and the majority of them are nouns (80,000+).

The ImageNet dataset used for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in this work contains 1,261,406 training images, 50,000 validation images, and 150,000 test images. There are 1000 synsets, and the number of images for each synset ranges from 668 to 3047. The 50,000 validation images are divided into 50 images per synset. The steps to install the ImageNet dataset on Shaheen-II are presented in the supplemental. Briefly, the file quota must be increased to allow the user to store around 1.2 million images. Then, a script in the dataset package installation prepares the training and test directories for the images. Finally, as per the recommendation to use the dataset, some images are patched using a script to avoid training and test errors.

The main stages of parallel training of a DNN are detailed in supplemental, Section 2. All the processes call the MPI_Allreduce collective routine during Step 3 of an epoch. Each process passes a message, a vector of gradients of size m bytes, to the MPI_Allreduce collective routine to obtain the same global vector of average gradients from the input vectors of gradients.

We follow a strict statistical methodology to ensure the experimental results are reliable. For each data point, the experiment is repeated until the sample mean lies in the 95% confidence interval and a precision of 0.025 (2.5%) is achieved. For this purpose, Student's t-test is used, assuming that the individual observations are independent and their population follows the normal distribution. The validity of these assumptions is verified by plotting the distributions of observations and using Pearson's Test.

The inputs to SUARA2 are the number of processes, P; the message size, m; the message segment size, m_s; the time of computation per byte, γ; and a (α, β) pair for each Open MPI allreduce algorithm. Each process contains a local vector of values (size equal to m bytes) that is reduced. In our experiments, the message segment size, m_s, is set to 8 KB, which is also commonly used in Open MPI.

The parameter γ is algorithm-independent and is estimated using separate communication experiments. The Hockney model parameters α_a and β_a specific to each allreduce algorithm, a, and used in the Open MPI analytical models of the allreduce collective algorithms are estimated using the best practices outlined in [36], [30]. For each allreduce algorithm a, a separate communication experiment is designed so that the algorithm itself would be involved in the execution of the experiment. Moreover, the execution time of this experiment is dominated by the execution time of this allreduce algorithm. Therefore, each allreduce algorithm a is executed by employing diverse sets of P and m and the execution times are measured. Then, a system of linear equations with α_a and β_a as unknowns is derived from these experiments. Finally, linear regression is applied to find α_a and β_a. The values of $(\alpha_{rs}, \beta_{rs})$ obtained for the allreduce algorithm, *ring with segmentation*, on Shaheen-II are given in Table 4.

For a given P, the optimal Open MPI allreduce algorithm in the process rows and columns can vary depending on the message size m. Fig. 6 shows the message sizes employed in MPI_Allreduce during an epoch in the DNN training on ImageNet ILSVRC2010

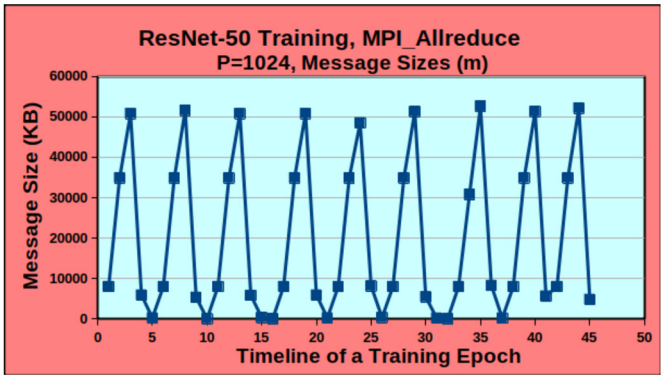


Fig. 6. A timeline fragment during a training epoch showing the message sizes employed in MPI_Allreduce calls of ResNet-50 DNN on ImageNet ILSVRC2010 dataset for $P = 1024$.

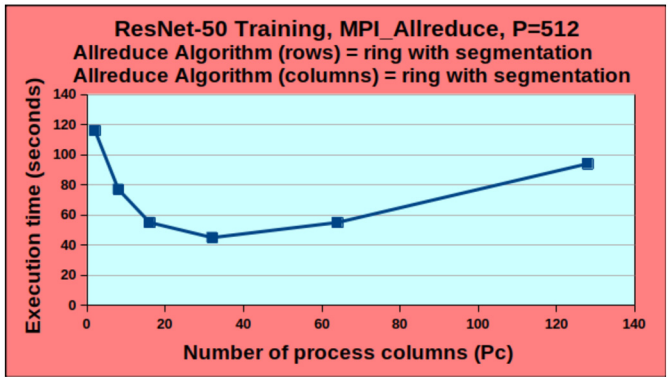


Fig. 7. Execution times per epoch of MPI_Allreduce for $P = 512$ employing *ring with segmentation* algorithm in the process rows and columns during ResNet-50 training using ImageNet. The x-axis shows the number of process columns (P_c) used in the 2D process grid arrangement, $(\frac{P}{P_c}, P_c)$. The optimal value of P_c is 32.

dataset. The message sizes range from 4 KB to 52 MB. For message sizes less than or equal to 1 MB, the SUARA2 Open MPI wrapper invokes the best native Open MPI algorithm to fulfil the allreduce operation. For message sizes greater than 1 MB, the SUARA2 Open MPI wrapper employs the Algorithm 1 described in the Section, "SUARA2: Pseudocode." The algorithm finds the optimal allreduce combination from thirty-six allreduce combinations using the $SUARA2\_find\_optimals$ function. An allreduce combination is a pair of identifiers representing the allreduce algorithms employed in the process rows and columns.

Fig. 7 shows for different (P_r, P_c) combinations of $P = 512$, the execution time per epoch of SUARA2 during the ResNet-50 training. The allreduce algorithm employed in the process rows and the columns is *ring with segmentation*. One can see that the optimal process grid arrangement is $(\frac{P}{P_c}, P_c)$ where P_c is the best integer approximation to $\sqrt{P}$ confirming the results from our theoretical analysis.

We check the execution time of SUARA2 with the actual experimental times for the thirty-six different allreduce combinations for representative sets of message sizes (m) and the number of processes (P) on our experimental platform. We observe that the optimal allreduce combination determined by SUARA2 is always the same as the best allreduce combination out of the different allreduce combinations that are experimentally evaluated. We also observe through our experiments that SUARA2 outperforms allreduce algorithms employing 2-step decomposition (reduce_scatter, allgather) and 3-step decomposition (reduce_scatter, allreduce, allgather) for a representative range of large message sizes and P ($P \leq 1024$) on the Shaheen-II supercomputer (Cray CX40).

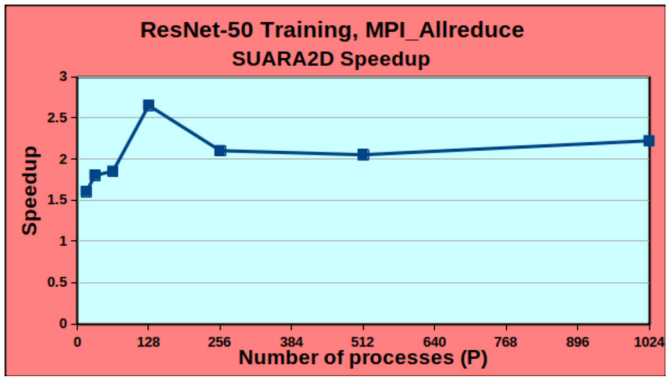


Fig. 8. The speedup of SUARA2 over the native Open MPI allreduce routine against the number of processes (P).

Fig. 8 shows the speedup of SUARA2 over the native Open MPI allreduce routine for the values of P, {16, 32, 64, 128, 256, 512, 1024}. The speedup is the ratio of the execution time of the native Open MPI allreduce routine divided by the execution time of SUARA2. For $P \leq 64$, the allreduce algorithm employed in the process rows and columns is *ring without segmentation*. For $64 < P \leq 1024$, the allreduce algorithm employed in the process rows and columns is *ring with segmentation*. In addition, the optimal 2D process grid arrangement is $(\frac{P}{P_c}, P_c)$, where P_c is the best integer approximation to $\sqrt{P}$. The average and maximum speedups are 2x and 2.65x. One can observe that the speedup graph displays the trend that asymptotically (for large P) approaches the behaviour of $\sqrt{P}$ validating our theoretical analysis.

We observe that the optimal allreduce combination found by SUARA2 has the same allreduce algorithm in the process rows and columns in all our experiments employing message sizes, $m \in \{4\text{ KB}, ..., 52\text{ MB}\}$, and $P \in \{1, ..., 1024\}$. However, in theory, the optimal allreduce algorithms employed in the process rows and columns can be different.

The maximum reduction of training times of ResNet-50 DNN on the ImageNet ILSVRC2010 dataset is 9% for $P = 1024$ processes. This experimental finding was the best we could accomplish given the limited access and duration of our experiments on Shaheen-II, which narrowed their extent and scope.

6. Conclusion

Deep neural networks (DNNs) have fuelled impressive innovations in computer vision, speech recognition, natural language processing, bioinformatics, drug design, medical image analysis, and climate science. However, accelerating the training of DNNs is a formidable challenge that is overcome by parallelization strategies. The efficiency of parallel deep learning packages employing synchronous stochastic gradient descent relies crucially on the performance of MPI allreduce collective communication operation.

MPI implementations provide many algorithms for the allreduce collective routine that can be selected using four approaches, exhaustive experimentation, empirical automatic selection, implementation-independent analytical performance modelling, and implementation-aware analytical performance modelling combined with accurate model parameter estimation. State-of-the-art platform-independent methods for performance optimization of the allreduce collective communication employ either functional decomposition of the global allreduce operation into sub-operations that are not allreduce operations or message decomposition/segmentation.

In this work, we proposed SUARA, a novel scalable universal allreduce meta-algorithm comprising L serial steps executed by P MPI processes. At each step, SUARA partitions the set of processes

into subsets that execute optimally selected library algorithms to solve sub-allreduce problems on these subsets in parallel to accomplish the whole allreduce operation after completing all the L steps.

Therefore, SUARA is a platform-independent multi-step allreduce meta-algorithm employing process decomposition to optimize the global allreduce operation using native allreduce algorithms as sub-operations. Thus, it differs from the state-of-the-art platform-independent allreduce algorithms in two respects. First, it is based on process decomposition, not functional or message decomposition. Second, it only employs sub-allreduce operations that execute optimally selected native allreduce algorithms.

We proved that the processes executing SUARA must naturally form a L-dimensional rectangular arrangement for maximum parallelism and to ensure the correctness of the allreduce operation.

We then designed, theoretically studied and implemented a two-step SUARA called SUARA2 on top of the Open MPI library. We use the traditional calculus approach to determine the optimal 2D process arrangement for each allreduce combination employing different native allreduce algorithms in process rows and columns. SUARA2 then selects the optimal allreduce combination out of the thirty-six allreduce combinations using the optimal 2D process arrangement determined for each allreduce combination and employs this optimal combination for the execution of the MPI allreduce collective operation.

We proved that SUARA2 exhibits a theoretical asymptotic speedup $\mathcal{O}(\sqrt{P})$ over the best Open MPI allreduce algorithm. Furthermore, we demonstrated the practical efficiency of SUARA2 by accelerating ResNet-50 deep neural network training by 9% on the ImageNet dataset on Shaheen-II supercomputer employing 1024 nodes. SUARA2 exhibited an average speedup of over 2x over the native Open MPI allreduce routine (the maximum being 2.65x).

In our future work, we will pursue two research directions. The first research direction will look into the design and implementation of SUARA, comprising more than two steps. The second research direction will focus on the efficiency of the optimized routine for training ResNet-50 DNN on GPUs employing the NCCL communication library.

Declaration of competing interest

Alexey L. Lastovetsky reports financial support was provided by Science Foundation Ireland. Ravi Reddy Manumachu reports financial support was provided by Sustainable Energy Authority of Ireland.

Data availability

Data will be made available on request.

Acknowledgment

For computer time, this research used the resources of the Supercomputing Laboratory at King Abdullah University of Science & Technology (KAUST) in Thuwal, Saudi Arabia.

Appendix A. Supplementary material

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jpdc.2023.104767.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, TensorFlow: a system for large-scale machine learning, in: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, USENIX Association, 2016, pp. 265–283.

[2] G. Almási, P. Heidelberger, C.J. Archer, X. Martorell, C.C. Erway, J.E. Moreira, B. Steinmacher-Burow, Y. Zheng, Optimization of MPI collective communication on BlueGene/L systems, in: Proceedings of the 19th Annual International Conference on Supercomputing, ICS'05, ACM, 2005, pp. 253–262.

[3] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, J. Chen, J. Chen, Z. Chen, M. Chrzanowski, A. Coates, G. Diamos, K. Ding, N. Du, E. Elsen, J. Engel, W. Fang, L. Fan, C. Fougner, L. Gao, C. Gong, A. Hannun, T. Han, L.V. Johannes, B. Jiang, C. Ju, B. Jun, P. LeGresley, L. Lin, J. Liu, Y. Liu, W. Li, X. Li, D. Ma, S. Narang, A. Ng, S. Ozair, Y. Peng, R. Prenger, S. Qian, Z. Quan, J. Raiman, V. Rao, S. Satheesh, D. Seetapun, S. Sengupta, K. Srinet, A. Sriram, H. Tang, L. Tang, C. Wang, J. Wang, K. Wang, Y. Wang, Z. Wang, Z. Wang, S. Wu, L. Wei, B. Xiao, W. Xie, Y. Xie, D. Yogatama, B. Yuan, J. Zhan, Z. Zhu, Deep speech 2: end-to-end speech recognition in English and Mandarin, in: Proceedings of the 33rd International Conference on International Conference on Machine Learning – vol. 48, ICML'16, 2016, pp. 173–182, JMLR.org.

[4] M. Bayatpour, J. Maqbool Hashmi, S. Chakraborty, H. Subramoni, P. Kousha, D.K. Panda, SALaR: scalable and adaptive designs for large message reduction collectives, in: Proceedings of the IEEE International Conference on Cluster Computing, 2018, pp. 12–23.

[5] T. Ben-Nun, T. Hoefler, Demystifying parallel and distributed deep learning: an in-depth concurrency analysis, ACM Comput. Surv. 52 (4) (Aug. 2019).

[6] M. Bojarski, D.D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L.D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, K. Zieba, End to end learning for self-driving cars, CoRR, arXiv:1604.07316 [abs], 2016.

[7] A. Castelló, E.S. Quintana-Ortí, J. Duato, Accelerating distributed deep neural network training with pipelined MPI allreduce, Clust. Comput. 24 (2021) 1–17.

[8] A. Castelló, M. Catalán, M.F. Dolz, E.S. Quintana-Ortí, J. Duato, Analyzing the impact of the MPI allreduce in distributed training of convolutional neural networks, Computing 105 (5) (2023) 1101–1119.

[9] E.W. Chan, M.F. Heimlich, A. Purkayastha, R.A. Van De Geijn, On optimizing collective communication, in: Proceedings of the 2004 IEEE International Conference on Cluster Computing, IEEE, 2004, pp. 145–155.

[10] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, Z. Zhang, MXNet: a flexible and efficient machine learning library for heterogeneous distributed systems, CoRR, arXiv:1512.01274 [abs], 2015.

[11] T. Chilimbi, Y. Suzue, J. Apacible, K. Kalyanaraman, Project adam: building an efficient and scalable deep learning training system, in: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, USENIX Association, 2014, pp. 571–582.

[12] D.C. Cireşan, A. Giusti, L.M. Gambardella, J. Schmidhuber, Mitosis detection in breast cancer histology images with deep neural networks, in: K. Mori, I. Sakuma, Y. Sato, C. Barillot, N. Navab (Eds.), Medical Image Computing and Computer-Assisted Intervention, MICCAI 2013, Springer Berlin Heidelberg, 2013, pp. 411–418.

[13] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, P. Kuksa, Natural language processing (almost) from scratch, J. Mach. Learn. Res. 12 (2011) 2493–2537.

[14] E.D. Coninck, S. Bohez, S. Leroux, T. Verbelen, B. Vankeirsbilck, P. Simoens, B. Dhoedt, DIANNE: a modular framework for designing, training and deploying deep neural networks on heterogeneous distributed infrastructure, J. Syst. Softw. 141 (2018) 52–65.

[15] J. Dean, G.S. Corrado, R. Monga, K. Chen, M. Devin, Q.V. Le, M.Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, A.Y. Ng, Large scale distributed deep networks, in: Proceedings of the 25th International Conference on Neural Information Processing Systems – vol. 1, NIPS'12, Curran Associates Inc., 2012, pp. 1223–1231.

[16] Facebook, Gloo: collective communications library, https://github.com/facebookincubator/gloo.git, 2020.

[17] P. Goyal, P. Dollár, R.B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, K. He, Accurate, large minibatch SGD: training imagenet in 1 hour, CoRR, arXiv:1706.02677 [abs], 2017.

[18] R.L. Graham, T.S. Woodall, J.M. Squyres, Open MPI: a flexible high performance MPI, in: Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics, PPAM'05, Springer-Verlag, 2005, pp. 228–239.

[19] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Comput. 22 (6) (1996) 789–828.

[20] R. Gupta, P. Balaji, D. Panda, J. Nieplocha, Efficient collective operations using remote memory operations on VIA-based clusters, in: Proceedings of the 17th International Parallel and Distributed Processing Symposium, IEEE Computer Society, 2003, 9 pp.

[21] Q. Ho, J. Cipar, H. Cui, J.K. Kim, S. Lee, P.B. Gibbons, G.A. Gibson, G.R. Ganger, E.P. Xing, More effective distributed ML via a stale synchronous parallel parameter server, in: Proceedings of the 26th International Conference on Neural Information Processing Systems – vol. 1, NIPS'13, Curran Associates Inc., 2013, pp. 1223–1231.

[22] R.W. Hockney, The communication challenge for MPP: Intel Paragon and Meiko CS-2, Parallel Comput. 20 (3) (1994) 389–398.

[23] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G.R. Ganger, P.B. Gibbons, O. Mutlu, Gaia: geo-distributed machine learning approaching LAN speeds, in: Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation, NSDI'17, USENIX Association, 2017, pp. 629–647.
[24] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, S. Shi, X. Chu, Highly scalable deep learning training system with mixed-precision: training ImageNet in four minutes, CoRR, arXiv:1807.11205 [abs], 2018.
[25] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, Caffe: convolutional architecture for fast feature embedding, in: Proceedings of the 22nd ACM International Conference on Multimedia, MM'14, ACM, 2014, pp. 675–678.
[26] KAUST Supercomputing Lab, Specification of Cray XC40 Shaheen-II, https://www.hpc.kaust.edu.sa/content/shaheen-ii, 2022.
[27] A. Mamidala, J. Liu, D. Panda, Efficient barrier and allreduce on infiniband clusters using multicast and adaptive algorithms, in: Proceedings of IEEE International Conference on Cluster Computing (IEEE Cat. No. 04EX935), IEEE Computer Society, 2004, pp. 135–144.
[28] A.R. Mamidala, G. Kollias, C. Ward, F. Artico, MXNET-MPI: embedding MPI parallelism in parameter server task model for scaling deep learning, CoRR, arXiv:1801.03855 [abs], 2018.
[29] Message Passing Interface Forum, MPI: A message-passing interface standard, https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/mpi-report.html, 1995.
[30] E. Nuriyev, A. Lastovetsky, Efficient and accurate selection of optimal collective communication algorithms using analytical performance modeling, IEEE Access 9 (2021) 109355–109373.
[31] E. Nuriyev, J.-A. Rico-Gallego, A. Lastovetsky, Model-based selection of optimal MPI broadcast algorithms for multi-core clusters, J. Parallel Distrib. Comput. 165 (2022) 1–16.
[32] NVIDIA, NVIDIA collective communications library, https://developer.nvidia.com/nccl, 2020.
[33] P. Patarasuk, X. Yuan, Bandwidth optimal all-reduce algorithms for clusters of workstations, J. Parallel Distrib. Comput. 69 (2) (2009) 117–124.
[34] J. Pjesivac-Grbovic, Towards automatic and adaptive optimizations of MPI collective operations, Ph.D. thesis, University of Tennessee, Knoxville, 2007.
[35] R. Rabenseifner, Optimization of collective reduction operations, in: M. Bubak, G.D. van Albada, P.M.A. Sloot, J. Dongarra (Eds.), Computational Science, ICCS 2004, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 1–9.
[36] J.A. Rico-Gallego, J.C. Díaz-Martín, R.R. Manumachu, A.L. Lastovetsky, A survey of communication performance models for high-performance computing, ACM Comput. Surv. 51 (6) (jan 2019).
[37] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A.C. Berg, L. Fei-Fei, ImageNet large scale visual recognition challenge, Int. J. Comput. Vis. 115 (3) (2015) 211–252.
[38] F. Seide, A. Agarwal, CNTK: Microsoft's open-source deep-learning toolkit, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'16, ACM, 2016, p. 2135.
[39] A. Sergeev, M.D. Balso, Horovod: fast and easy distributed deep learning in tensorflow, CoRR, arXiv:1802.05799 [abs], 2018.
[40] T. Thao Nguyen, M. Wahib, R. Takano, Efficient MPI-allreduce for large-scale deep learning on GPU-clusters, Concurr. Comput. 33 (12) (2021) e5574.
[41] V. Tipparaju, J. Nieplocha, D. Panda, Fast collective operations using shared and remote memory access protocols on clusters, in: Proceedings of the 17th International Parallel and Distributed Processing Symposium, IEEE Computer Society, 2003, 10 pp.
[42] R. Vandegeijn, On global combine operations, J. Parallel Distrib. Comput. 22 (2) (1994) 324–328.
[43] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, J.S. Rellermeyer, A survey on distributed machine learning, ACM Comput. Surv. 53 (2) (Mar. 2020).
[44] J. Worringen, Pipelining and overlapping for MPI collective operations, in: Proceedings of the 28th Annual IEEE International Conference on Local Computer Networks, 2003, pp. 548–557.
[45] E.P. Xing, Q. Ho, W. Dai, J.K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, Y. Yu, Petuum: a new platform for distributed machine learning on big data, IEEE Trans. Big Data 1 (2) (2015) 49–67.
[46] Y. Zhao, L. Wang, W. Wu, G. Bosilca, R. Vuduc, J. Ye, W. Tang, Z. Xu, Efficient communications in training large scale neural networks, in: Proceedings of the on Thematic Workshops of ACM Multimedia 2017, Thematic Workshops'17, ACM, 2017, pp. 110–116.

Emin Nuriyev received a PhD degree from the School of Computer Science, University College Dublin in 2021. He received BSc and MSc degrees in Applied Mathematics from the Baku State University in 2005 and 2007 respectively. His main research interests include algorithms and models for High-Performance Computing.

Ravi Reddy Manumachu received a B.Tech degree from I.I.T, Madras in 1997 and a PhD degree from the School of Computer Science, University College Dublin in 2005. He is currently an assistant professor in the School of Computer Science, University College Dublin. His main research interests include high performance heterogeneous computing and energy-efficient computing.

Samar Aseeri is a computational scientist with over 10 years of experience in the field of high performance computing (HPC). She is currently a member of the Extreme Computing Center at KAUST, where she works on developing and using HPC resources to solve complex scientific problems. Her PhD is in Applied Mathematics from the Umm Al-Qurra University, Makkah in 2009. She spent a year and a half at IBM's T.J. Watson Research Center, where she worked on HPC applications for a variety of scientific domains.

Dr. Aseeri is an active member of the HPC community. She has published over 10 papers in top academic journals and conferences, and she is a frequent speaker at HPC events. She is also a member of the IEEE and the ACM.

Mahendra Verma received his Ph.D. degree from University of Maryland, College Park. Presently, he is a Professor at the Physics Department of IIT Kanpur, India. He is a recipient of Swarnajayanti fellowship, and author of an introductory book "Introduction to Mechanics". Mahendra's research interests include turbulence, nonlinear dynamics, and high-performance computing. He is a lead developer of the spectral code TARANG that can simulate variety of fluid flows including fluids, magnetohydrodynamics, and thermal convection.

Alexey L. Lastovetsky received a Ph.D. degree from the Moscow Aviation Institute in 1986, and a Doctor of Science degree from the Russian Academy of Sciences in 1997. His main research interests include algorithms, models, and programming tools for high performance heterogeneous computing. He is currently Associate Professor in the School of Computer Science at University College Dublin (UCD). At UCD, he is also the founding Director of the Heterogeneous Computing Laboratory.