

Dhabaleswar K. Panda
Michael Sullivan (Eds.)

LNCS 13214

Supercomputing Frontiers

7th Asian Conference, SCFA 2022
Singapore, March 1–3, 2022
Proceedings

 Springer

OPEN ACCESS

Founding Editors

Gerhard Goos

Karlsruhe Institute of Technology, Karlsruhe, Germany

Juris Hartmanis

Cornell University, Ithaca, NY, USA

Editorial Board Members

Elisa Bertino

Purdue University, West Lafayette, IN, USA

Wen Gao

Peking University, Beijing, China

Bernhard Steffen 

TU Dortmund University, Dortmund, Germany

Moti Yung 

Columbia University, New York, NY, USA

More information about this series at <https://link.springer.com/bookseries/558>

Dhabaleswar K. Panda · Michael Sullivan (Eds.)

Supercomputing Frontiers

7th Asian Conference, SCFA 2022

Singapore, March 1–3, 2022

Proceedings

Editors

Dhabaleswar K. Panda
Department of Computer Science
and Engineering
The Ohio State University
Columbus, OH, USA

Michael Sullivan
Material Science and Chemistry
A*STAR Institute of High Performance
Computing
Singapore, Singapore



ISSN 0302-9743

ISSN 1611-3349 (electronic)

Lecture Notes in Computer Science

ISBN 978-3-031-10418-3

ISBN 978-3-031-10419-0 (eBook)

<https://doi.org/10.1007/978-3-031-10419-0>

© The Editor(s) (if applicable) and The Author(s) 2022. This book is an open access publication.

Open Access This book is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this book are included in the book's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the book's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

As the share of supercomputers in Asia continues to increase, the relevance of supercomputing merits a supercomputing conference for Asia. Supercomputing Asia 2022 (SCA22) was an umbrella of notable supercomputing events that promoted a vibrant HPC ecosystem in Asian countries.

The technical program of SCA22 provided a platform for leaders from both academia and industry to interact and to discuss visionary ideas, important global trends, and substantial innovations in supercomputing. Called the Supercomputing Frontiers Asia (SCFA) technical paper program, SCFA22, consisted of four tracks:

- Application, Algorithms, and Libraries
- Architecture, Network/Communications, and Management
- Data, Storage, and Visualization
- Programming and Systems Software

The submitted papers for the technical papers program went through a rigorous peer review process by an international program committee. A set of eight papers were finally selected for inclusion in the proceedings. The accepted papers cover a range of topics including HPC communication, GPU programming models, network service, large data set transfer, molecular dynamics simulation, and large-scale parallel applications. The quality of the papers is reflected in the proceedings that you find here. We would like to thank all authors for their submissions to this conference. Our sincere thanks to all Program Committee members for doing high-quality and in-depth reviewing of the submissions and selecting the papers for this year's program. We would like to thank the conference organizers for giving us the opportunity to serve this year's conference as the Co-Chairs for SCFA22.

April 2022

Dhabaleswar K. Panda
Michael Sullivan

Organization

Program Co-chairs

Dhableswar K. (DK) Panda
Michael Sullivan

The Ohio State University, USA
A*STAR Institute of High Performance
Computing, Singapore

Program Committee

Ritu Arora
Olivier Aumage
Ron Brightwell
Rajkumar Buyya
Maciej Cytowski
Sandra Gesing

The University of Texas at San Antonio, USA
Inria, France
Sandia National Laboratories, USA
The University of Melbourne, Australia
Pawsey Supercomputing Centre, Australia
University of Illinois Discovery Partners Institute,
USA

Mark Gray
Bilel Hadri
John Kim

Pawsey Supercomputing Centre, Australia
KAUST Supercomputing Lab, Saudi Arabia
Korea Advanced Institute of Science and
Technology, South Korea

Kishore Kothapalli

International Institute of Information Technology,
Hyderabad, India

Bu Sung Lee
Fang-Pang Lin

Nanyang Technological University, Singapore
National Center for High-Performance
Computing, Taiwan

Jing Lou

A*STAR Institute of High-Performance
Computing, Singapore

Piotr Luszczek
George S. Markomanolis
Ronald Minnick
Antonio J. Peña
Depei Qian
Martin Schulz
Ryota Shioya
Nathan Tallent
Michela Taufer

University of Tennessee Knoxville, USA
CSC - IT Center for Science Ltd., Finland
Sandia National Laboratory, USA
Barcelona Supercomputing Center (BSC), Spain
Beihang University Member, China
Technical University of Munich, Germany
The University of Tokyo, Japan
Pacific Northwest National Laboratory, USA
University of Tennessee, Knoxville, USA

Jingbo Wang
Jianfeng Zhan

Joey Tianyi Zhou

The Australian National University, Australia
Institute of Computing Technology,
Chinese Academy of Sciences, China
A*STAR Institute of High Performance
Computing, Singapore

Contents

High Performance Parallel LOBPCG Method for Large Hamiltonian Derived from Hubbard Model on Multi-GPU Systems	1
<i>Susumu Yamada, Toshiyuki Imamura, and Masahiko Machida</i>	
Vapor Condensation Under Electric Field: A Study Using Molecular Dynamics Simulation	20
<i>Pengyu Wang and Zhong Chen</i>	
The Effect of Wing Mass and Wing Elevation Motion During Insect Forward Flight	31
<i>Jie Yao and K. S. Yeo</i>	
Exploring the Dynamics of Quantum Information in Many-Body Localised Systems with High Performance Computing	43
<i>Shao-Hen Chiew, Leong-Chuan Kwek, and Chee-Kong Lee</i>	
On the Difference Between Shared Memory and Shared Address Space in HPC Communication	59
<i>Atsushi Hori, Kaiming Ouyang, Balazs Gerofi, and Yutaka Ishikawa</i>	
Evaluating GPU Programming Models for the LUMI Supercomputer	79
<i>George S. Markomanolis, Aksel Alpay, Jeffrey Young, Michael Klemm, Nicholas Malaya, Aniello Esposito, Jussi Heikonen, Sergei Bastrakov, Alexander Debus, Thomas Kluge, Klaus Steiniger, Jan Stephan, Rene Widera, and Michael Bussmann</i>	
Evaluating Methods of Transferring Large Datasets	102
<i>Jakub Kopeć</i>	
Service Function Chaining Design & Implementation Using Network Service Mesh in Kubernetes	121
<i>Abdullah Bittar, Ziqiang Wang, Amir Aghasharif, Changcheng Huang, Gauravdeep Shami, Marc Lyonais, and Rodney Wilson</i>	
Author Index	141



High Performance Parallel LOBPCG Method for Large Hamiltonian Derived from Hubbard Model on Multi-GPU Systems

Susumu Yamada^{1(✉)}, Toshiyuki Imamura², and Masahiko Machida¹

¹ Japan Atomic Energy Agency, Kashiwa, Chiba, Japan
yamada.susumu@jaea.go.jp

² RIKEN, Kobe, Hyogo, Japan

Abstract. The physical property of the Hubbard model can be understood by solving the eigenvalue problem for the Hamiltonian derived from the model. Since the Hamiltonian is a large sparse matrix, an iteration method is usually utilized for solving the problems. One of effective solvers for this problem is the LOBPCG (Locally Optimal Block Preconditioned Conjugate Gradient) method. The tuning strategies of the method on GPU systems when all iteration vectors are stored in device memory have been proposed. In this research, we propose tuning strategies for parallel LOBPCG method on multi-GPU system when the Hamiltonian is large and some iteration vectors are stored in host memory. When the LOBPCG method is used for solving multi eigenpairs (eigenvalues and the corresponding eigenvectors), the number of iteration vectors, whose size is the same as the dimension of the Hamiltonian, is proportional to the number of the eigenpairs. On the other hand, the memory consumption for the non-zero elements of the Hamiltonian can be significantly reduced by considering the regular arrangement of the elements. Therefore, when we execute the LOBPCG method for a large Hamiltonian on GPUs, some of the vectors have to be stored on host memory and have to be transferred between host and device memory as needed. Since the cost of the data transfer is very large, we also propose the optimization for it. The simulation result on a multi-GPU system shows that the optimization of the data transfer is very effective for high performance computing.

Keywords: LOBPCG method · Multi-GPU systems · Data transfer between CPU and GPU

1 Introduction

Eigenvalue problems appear in a variety of fields such as quantum dynamics, structure analysis and economics. Therefore, many solvers for them have been

developed and the strategies to improve their performance have also been proposed. In the quantum dynamics, when we solve eigenvalue problems derived from quantum models, we can recognize quantum states which indicate properties of the models. In this research, we focus on the eigenvalue problem for the Hamiltonian derived from the Hubbard model and will propose the strategies to realize a high performance solver on multi-GPU systems. The model can exhibit the property of many interesting phenomena such as high-temperature superconductivity [9,11], therefore, a lot of physicists take interest in it. The Hamiltonian, which represents the energy of the Hubbard model, is given as

$$H = -t \sum_{i,j,\sigma} c_{j\sigma}^\dagger c_{i\sigma} + \sum_i U_i n_{i\uparrow} n_{i\downarrow}, \quad (1)$$

where t is the hopping parameter from a site to another one and U_i is the repulsive energy for one-site double occupation of two fermion the i -th site. Quantities $c_{i,\sigma}$, $c_{i,\sigma}^\dagger$ and $n_{i,\sigma}$ are the annihilation, the creation, and the number operator of a fermion with pseudo-spin σ on the i -th site, respectively. When we solve the ground state (the smallest eigenvalue and the corresponding eigenvector) of the Hamiltonian, we can understand the property of the model. Moreover, when we solve multi eigenpairs (eigenvalues and the corresponding eigenvectors), we can reveal more detail property. Therefore, many methods to solve the model have been proposed. One of the most accurate solvers is the exact diagonalization method which directly solves some eigenpairs of the Hamiltonian derived exactly from the model. At this time, the Hamiltonian becomes a huge sparse symmetric matrix. Accordingly, an iteration method, such as the Lanczos method and LOBPCG (Locally Optimal Block Preconditioned Conjugate Gradient) method [7,8], is usually utilized. And then, the parallelization strategies for multi-CPU systems [14] and the tuning ones for single-GPU systems [1,12,15,17] have been proposed. In this research, in order to realize the LOBPCG method for solving some eigenpairs using multi-GPU systems, we will propose the parallelization and tuning strategies. The parallelization not only realizes speedup by distributing the calculations, but also enables simulations for larger models by distributing data. Since the memory size of GPU is generally smaller than that of CPU, we can calculate larger models by storing data in CPU memory (host memory) than in only GPU memory (device memory). Accordingly, in order to simulate a larger model, we transfer the data that are required for an operation from host to device memory and temporarily store them in device memory, and then, we execute the operation using them. Moreover, we have to transfer the calculation results from device to host memory, if necessary. Since the speed of the data transfer between host and device memory is much slower than that of data transfer in GPU, it is important to decrease the cost of data transfer between host and device memory for high performance parallel LOBPCG method on multi-GPU systems. In order to realize the decrease, we focus on the data transfer between host and device memory in this paper. Accordingly, although it may be possible to perform faster using CPUs in addition to GPUs, we target the LOBPCG method whose all time-consuming operations are performed on only GPUs.

The rest of the paper of structured as follows. Section 2 presents the implement schemes on multi-GPU systems. We propose the tuning strategies in view of data transfer between host and device memory in Sect. 3. Section 4 shows the result of numerical experiments on HPE SGI 8600 in Japan Atomic Energy Agency. A summary and conclusion are given in Sect. 5.

2 LOBPCG Method for Solving Multi Eigenpairs on Multi-GPU Systems

We can solve m eigenpairs of the matrix H using the LOBPCG method shown in Fig. 1¹. The method requires m matrix-vector multiplications and some linear operations using iteration vectors $\mathbf{x}_k^{(i)}$, $\mathbf{w}_k^{(i)}$, $\mathbf{p}_k^{(i)}$, $\mathcal{X}_k^{(i)}$, $\mathcal{W}_k^{(i)}$, and $\mathcal{P}_k^{(i)}$

$$\begin{aligned}
 &\mathbf{x}_0^{(i)} := \text{an initial guess}, \mathbf{p}_0^{(i)} := 0, i = 1, \dots, m \\
 &\mathbf{x}_0^{(i)} := \mathbf{x}_0^{(i)} / \|\mathbf{x}_0^{(i)}\|, i = 1, \dots, m \\
 &\mathcal{X}_0^{(i)} := A\mathbf{x}_0^{(i)}, \mathcal{P}_0^{(i)} := 0, i = 1, \dots, m \\
 &\mu_{-1}^{(i)} := (\mathbf{x}_0^{(i)}, \mathcal{X}_0^{(i)}), \mathbf{w}_0^{(i)} := \mathcal{X}_0^{(i)} - \mu_{-1}^{(i)}\mathbf{x}_0^{(i)}, i = 1, \dots, m \\
 &\text{do } k=0, \dots \text{ until convergence} \\
 &\mathcal{W}_k^{(i)} := H\mathbf{w}_k^{(i)}, i = 1, \dots, m \\
 &S_A := \{\mathbf{x}_k^{(1)}, \dots, \mathbf{x}_k^{(m)}, \mathbf{w}_k^{(1)}, \dots, \mathbf{w}_k^{(m)}, \mathbf{p}_k^{(1)}, \dots, \mathbf{p}_k^{(m)}\}^T \\
 &\{\mathcal{X}_k^{(1)}, \dots, \mathcal{X}_k^{(m)}, \mathcal{W}_k^{(1)}, \dots, \mathcal{W}_k^{(m)}, \mathcal{P}_k^{(1)}, \dots, \mathcal{P}_k^{(m)}\} \\
 &S_B := \{\mathbf{x}_k^{(1)}, \dots, \mathbf{x}_k^{(m)}, \mathbf{w}_k^{(1)}, \dots, \mathbf{w}_k^{(m)}, \mathbf{p}_k^{(1)}, \dots, \mathbf{p}_k^{(m)}\}^T \\
 &\{\mathbf{x}_k^{(1)}, \dots, \mathbf{x}_k^{(m)}, \mathbf{w}_k^{(1)}, \dots, \mathbf{w}_k^{(m)}, \mathbf{p}_k^{(1)}, \dots, \mathbf{p}_k^{(m)}\} \\
 &\text{Solve the generalized eigenvalue problem } S_A \mathbf{v} = \mu S_B \mathbf{v} \text{ to obtain the} \\
 &m \text{ smallest eigenvalues } \mu^{(i)} \text{ and the corresponding eigenvectors } \mathbf{v}^{(i)} = \\
 &(\alpha_1^{(i)}, \alpha_2^{(i)}, \dots, \alpha_m^{(i)}, \beta_1^{(i)}, \beta_2^{(i)}, \dots, \beta_m^{(i)}, \gamma_1^{(i)}, \gamma_2^{(i)}, \dots, \gamma_m^{(i)})^T (i = 1, \dots, m), \\
 &\mathbf{x}_{k+1}^{(i)} := \sum_{j=1}^m \{\alpha_j^{(i)} \mathbf{x}_k^{(j)} + \beta_j^{(i)} \mathbf{w}_k^{(j)} + \gamma_j^{(i)} \mathbf{p}_k^{(j)}\}, i = 1, \dots, m \\
 &\mathbf{p}_{k+1}^{(i)} := \sum_{j=1}^m \{\beta_j^{(i)} \mathbf{w}_k^{(j)} + \gamma_j^{(i)} \mathbf{p}_k^{(j)}\}, i = 1, \dots, m \\
 &\mathcal{X}_{k+1}^{(i)} := \sum_{j=1}^m \{\alpha_j^{(i)} \mathcal{X}_k^{(j)} + \beta_j^{(i)} \mathcal{W}_k^{(j)} + \gamma_j^{(i)} \mathcal{P}_k^{(j)}\}, i = 1, \dots, m \\
 &\mathcal{P}_{k+1}^{(i)} := \sum_{j=1}^m \{\beta_j^{(i)} \mathcal{W}_k^{(j)} + \gamma_j^{(i)} \mathcal{P}_k^{(j)}\}, i = 1, \dots, m \\
 &\mu_k^{(i)} := (\mathbf{x}_{k+1}^{(i)}, \mathcal{X}_{k+1}^{(i)}) / (\mathbf{x}_{k+1}^{(i)}, \mathbf{x}_{k+1}^{(i)}), i = 1, \dots, m \\
 &\mathbf{w}_{k+1}^{(i)} := T^{(i)-1} (\mathcal{X}_{k+1}^{(i)} - \mu_k^{(i)} \mathbf{x}_{k+1}^{(i)}), i = 1, \dots, m \\
 &\text{enddo}
 \end{aligned}$$

Fig. 1. LOBPCG method for solving the m smallest eigenvalues and the corresponding eigenvectors of a symmetric matrix H . $T^{(i)}$ is a preconditioner for the i -th smallest eigenvalues. And $\mathcal{X}_k^{(i)}$, $\mathcal{P}_k^{(i)}$, and $\mathcal{W}_k^{(i)}$ are $H\mathbf{x}_k^{(i)}$, $H\mathbf{p}_k^{(i)}$, and $H\mathbf{w}_k^{(i)}$, respectively. As convergence progresses, a set of iteration vectors $\{\mathbf{x}_k^{(1)}, \dots, \mathbf{x}_k^{(m)}, \mathbf{w}_k^{(1)}, \dots, \mathbf{w}_k^{(m)}, \mathbf{p}_k^{(1)}, \dots, \mathbf{p}_k^{(m)}\}$ becomes linearly dependent, and the general eigenvalue problem can not be solved. Therefore, in practice, we orthonormalize a set of the vectors to calculate the algorithm stably.

¹ In practice, in order to improve the convergence property, it is advisable to set the parameter m larger than the number of eigenpairs which we want to find.

($i = 1, 2, \dots, m$). Moreover, in order to execute the LOBPCG method stably, we have to orthonormalize iteration vectors $\mathbf{x}_k^{(i)}$, $\mathbf{w}_k^{(i)}$ and $\mathbf{p}_k^{(i)}$. At this time, S_B becomes the identity matrix. The parallelization schemes of the multiplication for the Hubbard model on multi-CPU systems have been proposed [14]. In addition, the tuning strategies for single-GPU systems have been also proposed [12, 15, 17]. We propose the parallelization of the multiplication on multi-GPU systems by combining the above two strategies appropriately. Since the size of device memory is typically a fraction of that of host memory, it is supposed in this paper that we store only the information of the matrix H and $2m$ vectors ($\mathbf{w}_k^{(i)}$ and $\mathbf{W}_k^{(i)}$, $i = 1, \dots, m$) in device memory and the other vectors ($\mathbf{x}_k^{(i)}$, $\mathbf{X}_k^{(i)}$, $\mathbf{p}_k^{(i)}$ and $\mathcal{P}_k^{(i)}$) in host memory. Here, the time-consuming operations of LOBPCG method are Hamiltonian matrix-vector multiplication operations and the vector operations (dot product, vector update and orthonormalization). In the following, we introduce the parallelization strategies in multi-GPU systems for each operation.

2.1 Hamiltonian Matrix-Vector Multiplications

When we solve m eigenpairs of the Hamiltonian, we have to operate m Hamiltonian-vector multiplications per iteration. Since each of these multiplications can be executed independently, we focus on parallelization and tuning strategies for one multiplication. The Hamiltonian is represented as

$$H = D + I_{\downarrow} \otimes A_{\uparrow} + A_{\downarrow} \otimes I_{\uparrow}, \quad (2)$$

where D is a diagonal matrix from the repulsive energy, and A_{\uparrow} (A_{\downarrow}) is a sparse symmetric matrix from the hopping of the up-spin (down-spin). And, I_{\uparrow} (I_{\downarrow}) is the identity matrix, dimension of which is the same as that of A_{\uparrow} (A_{\downarrow}). When the dimension of A_{\uparrow} (A_{\downarrow}) is n_{\uparrow} (n_{\downarrow}), the dimension of the Hamiltonian H is $n_{\uparrow} \times n_{\downarrow}$. Since the dimension of A_{\uparrow} and A_{\downarrow} is much smaller than that of H , we store the non-zero elements of A_{\uparrow} , A_{\downarrow} and D in device memory instead of the matrix H . Here, the multiplication of the Hamiltonian and vector is

$$Hv = Dv + (I_{\downarrow} \otimes A_{\uparrow})v + (A_{\downarrow} \otimes I_{\uparrow})v.$$

We transform the vector v to the matrix V as the following manner:

$$v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n_{\uparrow} \times n_{\downarrow}} \end{pmatrix} \rightarrow V = \begin{pmatrix} v_1 & v_{n_{\uparrow}+1} & \cdots & v_{n_{\uparrow} \times (n_{\downarrow}-1)+1} \\ v_2 & v_{n_{\uparrow}+2} & \cdots & v_{n_{\uparrow} \times (n_{\downarrow}-1)+2} \\ \vdots & \vdots & \vdots & \vdots \\ v_{n_{\uparrow}} & v_{2n_{\uparrow}} & \cdots & v_{n_{\uparrow} \times n_{\downarrow}} \end{pmatrix}, \quad (3)$$

and the diagonal elements of the matrix D to the matrix \bar{D} as the same manner. Here, the matrix-vector multiplication is change into the following matrix-matrix multiplication:

$$(HV)_{i,j} = \bar{D}_{i,j}V_{i,j} + \sum_k A_{\uparrow,i,k}V_{k,j} + \sum_k A_{\downarrow,j,k}V_{i,k},$$

where the subscript i, j of a matrix is the (i, j) -th element of the matrix. Since the matrix V is a dense matrix, we can execute the multiplications in parallel as follows:

CAL 1: $Y^c = \bar{D}^c \odot V^c + A_{\uparrow} V^c$,

COM 1: all-to-all communication from V^c to V^r ,

CAL 2: $Z^r = V^r A_{\downarrow}^T$,

COM 2: all-to-all communication from Z^r to Z^c ,

CAL 3: $Y^c = Y^c + Z^c$.

where the superscription c and r denotes the columnwise and rowwise partitioning in matrix data for the parallel calculation. And, \odot means an elementwise multiplication. The parallelization strategy requires two all-to-all communication operations per multiplication.

When all non-zero elements of the matrix H and the data of the decomposed matrices V^c and V^r , are stored in device memory, we can execute **CAL 1**, **CAL 2** and **CAL 3** by using the algorithms proposed for single-GPU systems [12, 15, 17]. Here, the storage layout of V^c and V^r should be row-major and column-major order, respectively, in order that $A_{\uparrow} V^c$ and $V^r A_{\downarrow}^T$ are performed with contiguous memory access. Therefore, the method requires the change of the storage layout of the matrix between column-major and row-major order, however, the operation can be executed with high performance by using the shared memory on GPU systems.

2.2 Vector Operations

The vector operations in the LOBPCG method can be categorized into the following three groups:

- Dot product for constructing $3m \times 3m$ -dimensional symmetric matrix (S_A in Fig. 1),
- Updating all column vectors of X_{k+1} , P_{k+1} , W_{k+1} , \mathcal{X}_{k+1} and \mathcal{P}_{k+1} , where $X_{k+1} = (\mathbf{x}_{k+1}^{(1)}, \dots, \mathbf{x}_{k+1}^{(m)})$, $W_{k+1} = (\mathbf{w}_{k+1}^{(1)}, \dots, \mathbf{w}_{k+1}^{(m)})$, $P_{k+1} = (\mathbf{p}_{k+1}^{(1)}, \dots, \mathbf{p}_{k+1}^{(m)})$, $\mathcal{X}_{k+1} = (\mathcal{X}_{k+1}^{(1)}, \dots, \mathcal{X}_{k+1}^{(m)}) (= HX_{k+1})$ and $\mathcal{P}_{k+1} = (\mathcal{P}_{k+1}^{(1)}, \dots, \mathcal{P}_{k+1}^{(m)}) (= HP_{k+1})$.
- Orthonormalization for all column vectors of X_{k+1} , P_{k+1} and W_{k+1} .

We discuss the dot product and update for vectors first, then the orthonormalization of vectors.

Dot Product and Vector Update. The parallelization of dot product can be realized by calculating the partial sum of dot product on each process and performing the sum-reduction for the partial sum by MPI_ALLREDUCE in all processes. The parallelization of vector update operation can be realized by performing the ‘axpy’ operation for the decomposed vectors on each process without data communication between processors. When all vectors are stored

```

dd=0.d0
do i=1,l
  do k=1,m
    stat=cudamemcpy(dx, x(n*(i-1)+1,k), n, cudamemcpyhosttodevice)
    stat=cudamemcpy(dp, p(n*(i-1)+1,k), n, cudamemcpyhosttodevice)
  enddo
  do k=1,m
    do j=1,m
      stat=cublasddot_v2(h, n, dx(1,k), 1, dp(1,j), 1, dd0(k,j))
    enddo
  enddo
  stat=cublasdaxpy_v2(h, m*m, 1.d0, 1, dd0, 1, dd,1)
enddo
dot0=dd
call mpi_allreduce(dot0,dot,..)

```

(a) dot product (x , p)

```

do i=1,l
  do k=1,m
    stat=cudamemcpy(dx, x(n*(i-1)+1,k), n, cudamemcpyhosttodevice)
    stat=cudamemcpy(dp, p(n*(i-1)+1,k), n, cudamemcpyhosttodevice)
  enddo
  dtmp=0.d0
  do k=1,m
    do j=1,m
      stat= cublasdaxpy_v2(h,n, $\beta_j^{(k)}$ ,w((n*(i-1)+1,j),1,dtmp(1,k),1)
      stat= cublasdaxpy_v2(h,n, $\gamma_j^{(k)}$ ,pd((1,j),1,dtmp(1,k),1)
    enddo
    stat=cudamemcpy(p(n*(i-1)+1,k), dtmp(1,k), n, cudamemcpydevicetohost)
  enddo
  do k=1,m
    do j=1,m
      stat= cublasdaxpy_v2(h,n, $\alpha_j^{(k)}$ ,xd((1,j),1,dtmp(1,k),1)
    enddo
    stat=cudamemcpy(x(n*(i-1)+1,k), dtmp(1,k), n, cudamemcpydevicetohost)
  enddo
enddo

```

(b) Vector update ($\mathbf{p}_k^{(i)}$ and $\mathbf{x}_k^{(i)}$ in Fig. 1.)

Fig. 2. Schematic CUDA Fortran codes of vector operations of the LOBPCG method for solving m eigenpairs on multi-GPU systems. Here, the number of tiles of the vectors is l and the dimension of each tiled vector is n . Therefore, the dimension of the decomposed vector on each process is $l \times n$. Here, h is cuBLAS library handle. And, x , p , dot and $dot0$ are stored in host memory and dx , dp , w , α , β , γ , dd , $dd0$ and $dtmp$ in device memory. The codes are simplified to indicate the relationship between the data transfer and the execution on a GPU. Therefore, the code should be extended appropriately to actually execute the LOBPCG method. Moreover, the data is transferred in one operation by packing data.

on device memory, the vector operations can be parallelized straightforwardly. However, in this research, 4 matrices (X_k , \mathcal{X}_k , P_k and \mathcal{P}_k) are stored in host memory, and the data have to be transferred to device memory. Therefore, we partition the vectors into some tiles and we transfer each tile of the vectors from host to device memory and execute the partial dot product operation on each tile (Fig. 2(a)) [10]. The vector update operation can be also executed using almost the same strategy (Fig. 2(b)). However, the operation requires transferring the updated vectors from device to host memory.

Orthonormalization of Vectors. Here, we discuss the orthonormalization of the iteration vectors $\mathbf{x}_k^{(i)}$, $\mathbf{p}_k^{(i)}$, and $\mathbf{w}_k^{(i)}$. In order that we execute the LOBPCG method for solving multi eigenpairs stably, a set of the basis of the subspace spanned by all iteration vectors, that is, a set of all column vectors of X_k , P_k , and W_k , should be linearly independent. Therefore, we should orthogonalize the basis per iteration. The orthogonalization can be realized by many methods such as the modified Gram-Schmidt (MGS) orthonormalization, TSQR, CholeskyQR, CholeskyQR2 and so on [2, 4, 13]. When we apply these methods for vectors directly, we have to transfer vectors stored in host memory. Therefore, we focus on the orthogonalization strategy for the LOBPCG method proposed by Hetmaniuk and Lehoueq (HL) [3, 5]. The HL strategy is as follows:

- Here, we represent the eigenvector corresponding to the i -th smallest eigenvalue of S_A as $\mathbf{v}^{(i)}$. And, it is assumed that a set of vectors $\{\mathbf{v}^{(1)}, \mathbf{v}^{(2)}, \dots, \mathbf{v}^{(m)}\}$ is orthonormal, moreover, a set of all column vectors of the matrices X_k , P_k and W_k in the k -th iteration is also orthonormal.
- When the (i, j) -th element of the matrices C^1 , C^2 and C^3 are defined as $C_{(i,j)}^1 = (\alpha_j^{(i)})$, $C_{(i,j)}^2 = (\beta_j^{(i)})$ and $C_{(i,j)}^3 = (\gamma_j^{(i)})$, that is,

$$\begin{pmatrix} C^1 \\ C^2 \\ C^3 \end{pmatrix} = (\mathbf{v}^{(1)}, \mathbf{v}^{(2)}, \dots, \mathbf{v}^{(m)}),$$

X_{k+1} and P_{k+1} are calculated by

$$(X_{k+1}, P_{k+1}) = (X_k, W_k, P_k)C, \quad C = \begin{pmatrix} C^1 & 0 \\ C^2 & C^2 \\ C^3 & C^3 \end{pmatrix}.$$

Here, a set of all column vectors of matrices X_{k+1} becomes orthonormal.

- We decompose matrix C into QR using QR decomposition. When we calculate X_{k+1} and P_{k+1} using the following formula

$$(X_{k+1}, P_{k+1}) = (X_k, W_k, P_k)Q,$$

all columns of X_{k+1} and P_{k+1} are orthonormal².

² X_{k+1} is the same as the above result, because a set of $(\mathbf{v}^{(1)}, \mathbf{v}^{(2)}, \dots, \mathbf{v}^{(m)})$ is orthonormal.

- Next, we orthonormalize the column vectors of W_{k+1} against those of X_{k+1} and P_{k+1} using the classical Gram-Schmidt (CGS) method, that is, W_{K+1} is updated by the following formula [6]

$$W_{k+1} = (I - X_{k+1}X_{k+1}^T - P_{k+1}P_{k+1}^T)W_{k+1}. \quad (4)$$

The method requires much less allreduce communication operations than the MGS method.

- Finally, we orthonormalize a set of the columns vectors of W_{k+1} using the MGS method. The MGS method requires a lot of allreduce communication operations, however, the number of the operations in this case is reduced by about one-ninth compared to the orthonormalization of all column vectors of three matrices X_{k+1} , P_{k+1} and W_{k+1} . Moreover, since we orthonormalize a set of only the column vectors of W_{k+1} stored in device memory, we do not need to transfer any vectors between host to device memory.

In this operation, when we find vectors to be a combination of other vectors, we eliminate them in this iteration. We show the schematic CUDA Fortran code for orthogonalizing the column vectors W_{k+1} against those of X_{k+1} and P_{k+1} in Fig. 3. The operation requires to transfer X_{k+1} and P_{k+1} from host to device memory twice.

Performance. Here, we examine the performance of these three operations using the above methods for solving the eigenvalue problems of the Hamiltonian derived from 5×4 -site Hubbard model with 5 up-spins and 5 down-ones using 16 GPUs of 4 nodes on HPE SGI8600 in Japan Atomic Energy Agency. The details of the system are shown in Table 1. Here, the dimension of the Hamiltonian is 240,374,016, therefore, the dimension of a partitioned vector for each GPU is 15,023,376.

Figure 4 shows the elapsed time of each operation for solving 5 or 10 eigenpairs. The result indicates that the elapsed times tend to increase as the number of tiles becomes larger. The reason is that as the number of the tiles increases, the number of data transfer operations increases and the data size for each transfer operation decreases, that is, the latency of data transfer increases and the throughput declines. Moreover, in this result, it is noted that the elapsed time of the vector update operation is unstable. The reason is that when we execute operations on four GPUs on the system whose node logic diagram as shown in Fig. 5, we generally run two processes on each processor. When two processes on a processor simultaneously transfer data between host and device memory in the same direction, the bus connected between a CPU and two GPUs are shared with two processes and the throughput per process is limited to about half of the throughput of data transfer using one process. In the beginning of the iteration, the data transfer operations on two processes of one processor are synchronized. However, when the number of the tiles is large, that is, the number of iterations for completing the operation is large, the data transfer operations in the same direction on the two processes gradually become out of synchronization and the

```

do i=1,l
  do k=1,m
    stat=cudamemcpy(dx, x(n*(i-1)+1,k), n, cudamemcpyhosttodevice)
    stat=cudamemcpy(dp, p(n*(i-1)+1,k), n, cudamemcpyhosttodevice)
  enddo
  do k=1,m
    do j=1,m
      stat=cublasddot_v2(h, n, dx(1,k), 1, w((n*(i-1)+1,j), 1, dd0(k,j))
      stat=cublasddot_v2(h, n, dp(1,k), 1, w((n*(i-1)+1,j), 1, dd1(k,j))
    enddo
  enddo
  stat=cublasdaxpy_v2(h, m*m, 1.d0, 1, dd0, 1, dh0,1)
  stat=cublasdaxpy_v2(h, m*m, 1.d0, 1, dd1, 1, dh1,1)
enddo
dot0=dh0
call mpi_allreduce(dot0,dot,...)
dd0=dot
dot0=dh1
call mpi_allreduce(dot0,dot,...)
dd1=dot
do i=1,l
  do k=1,m
    stat=cudamemcpy(dx, x(n*(i-1)+1,k), n, cudamemcpyhosttodevice)
    stat=cudamemcpy(px, p(n*(i-1)+1,k), n, cudamemcpyhosttodevice)
  enddo
  do k=1,m
    do j=1,m
      stat= cublasdaxpy_v2(h,n,dd0(k,j),dx(1,k),1,w((n*(i-1)+1,j),1)
      stat= cublasdaxpy_v2(h,n,dd1(k,j),dp(1,k),1,w((n*(i-1)+1,j),1)
    enddo
  enddo
enddo
enddo

```

Fig. 3. Schematic CUDA Fortran code of orthogonalizing the column vectors of W against those of X and P using the classical Gram-Schmidt method.

opposite directional transfer operations sometimes might be synchronized. They can be performed without conflict, and they achieve almost the same as the peak throughput when the size of the transferring data is large. Figure 6 shows their elapsed times for solving 10 eigenpairs. The result demonstrates that the elapsed time of the conventional method (synchronous data transfer) is in the interval between that of the same-directional data transfer with synchronization³ and that of the opposite-directional one (Fig. 7). And, it is confirmed that the interval of throughput for the two data transfer operations becomes narrow as the number of tiles increases. The reason is that when transferring large data, the

³ MPI.barrier operation is executed in the beginning of each iteration of the outermost loop in Fig. 2(b).

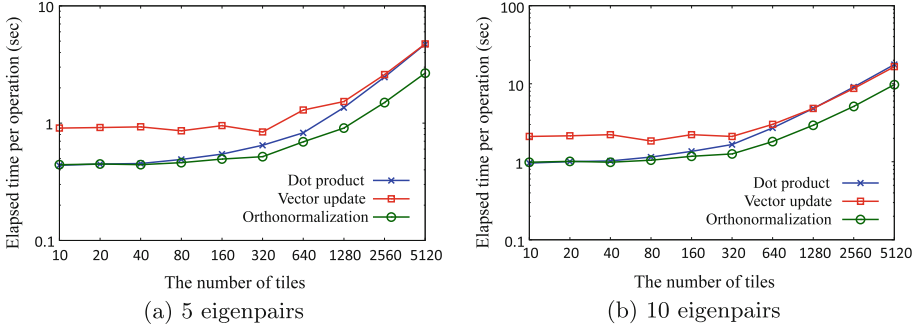


Fig. 4. Elapsed time of orthonormalization and dot product and vector update per operation.

Table 1. Details of GPU-system of HPE SGI 8600 in Japan Atomic Energy Agency.

CPU	Intel Xeon Gold 6248R (3.0 GHz, 35.75 MB cache) \times 2 CPUs
Number of CPUs per node	2
Number of cores per CPU	24
Memory of node	386 GB
GPU	NVIDIA Tesla V100 SXM2 32 GB memory
Number of GPUs per node	4
Connection between CPU-GPU	\times 16 PCIe (16 Gbytes/s)
Network	4 ports of $4 \times$ EDR Infini Band interfaces
Network Throughput	50 GB/s(12.5 GB/s \times 4) (one direction)

throughput of the opposite-directional data transfer is larger than that of the same-directional one, however, as the size of the transferring data per operation becomes small, that is, the number of tiles becomes larger, the former declines and ultimately becomes almost the same as the latter.

3 Optimization CPU-GPU Data Transfer

3.1 Asynchronous Data Transfer

When we execute the calculation consuming huge memory on a GPU, we have to transfer the necessary data from host to device memory (HtoD). The data transfer operation can be overlapped with the calculation on GPUs by using asynchronous data transfer. In actual, the dot product, the vector update and the orthonormalization operations shown in Sect. 2.2 can be overlapped with the data transfer using the multi-buffering strategy. Since the data transfer of the

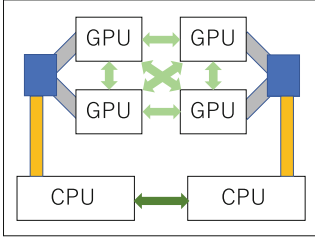


Fig. 5. Node logic diagram of GPU system on HPE SGI8600 in Japan Atomic Energy Agency.

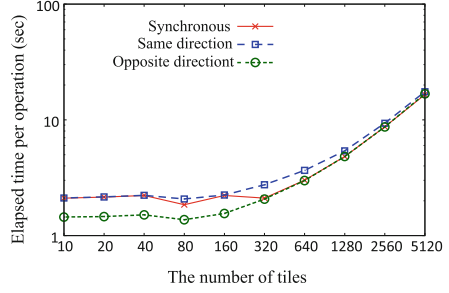


Fig. 6. Comparison of elapsed time of vector update operation using three types of data transfer.

```

call mpi_rank(rank,...)
:
do k=1,m
  if (mod(rank,2).eq.0) call mpi_barrier(...)
  HtoD data transfer
:
  Vector update operations
:
  if (mod(rank,2).eq.1) call mpi_barrier(...)
  DtoH data transfer
enddo

```

Fig. 7. Schematic code of vector update operation with performing HtoD data transfer on one process of a CPU and DtoH transfer on the other process at the same time. Here, two processes are run on each processor and the MPI ranks of these processes are set to be contiguous.

dot product and the orthonormalization operations is only HtoD, the overlap can be realized using the double-buffering. Moreover, the update vector operation requires to transfer the updated vectors from device to host memory (DtoH) in addition to HtoD data transfer. Therefore, the overlap of data transfer and calculation in the operation can be realized by using the triple-buffering.

Here, we compare the performance of the three operations using the synchronous data transfer with that using the asynchronous one. Figure 8 shows the relationship between the elapsed time of each operations and the number of the tiles. The results indicate that the method using the asynchronous data transfer is faster than the method using synchronous one for the dot product and the orthonormalization operations. On the other hand, for the vector update operation, when the number of tiles is small, the asynchronous data transfer realizes speedup, however, it is confirmed that the performance becomes unstable as the number of tiles increases. As a result, when the number is large, there are cases where the speedup effect can not be obtained.

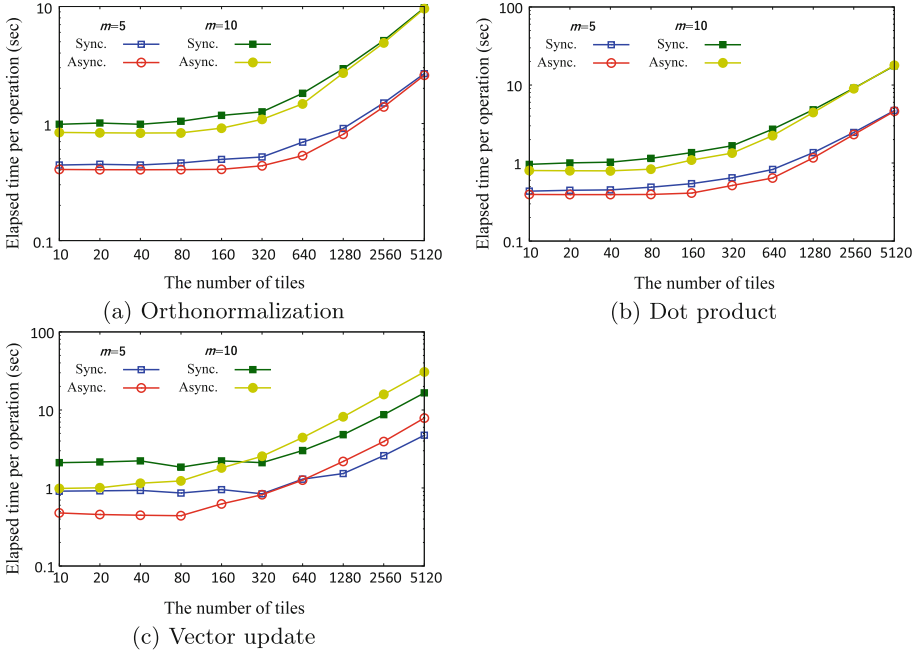


Fig. 8. Comparison of elapsed time using synchronous and asynchronous data transfer operations. Here, m means the number of eigenpairs to be solved.

3.2 Reduction of Data Transfers

After updating vectors, we orthonormalize the column vectors of W_{k+1} , and we calculate S_A using the dot product operations. The vector update operation requires HtoD transfer for four matrices X_k , \mathcal{X}_k , P_k and \mathcal{P}_k , and DtoH transfer for four updated matrices X_{k+1} , \mathcal{X}_{k+1} , P_{k+1} and \mathcal{P}_{k+1} . The column vectors of these four updated matrices will be used as is for the following calculations. On the other hand, the column vectors of W_{k+1} are the residual vectors, and they are modified by a preconditioning. Therefore, we can perform the dot product operations and the orthonormalization after applying preconditioner. However, when we apply a preconditioner which works elementwisely like point Jacobi preconditioner, we can modify the updated residual vectors elementwise. Accordingly, when we apply such a preconditioner or do not perform any preconditioning, we can calculate the partial sum of the dot products by using the subset of five matrices X_{k+1} , \mathcal{X}_{k+1} , W_{k+1} , P_{k+1} and \mathcal{P}_{k+1} before the data is transferred to the host memory⁴. Therefore, we can calculate $X_{k+1}W_{k+1}^T$ and $P_{k+1}W_{k+1}^T$ which are used for the orthogonalization of the column vectors of W_{k+1} against those of X_{k+1} and P_{k+1} without the HtoD data transfer. However, when we execute the orthogonalization using the result of the dot product, we have to transfer X_{k+1}

⁴ When we use the HL strategy, the column vectors of X_{k+1} and P_{k+1} are orthonormal. In this case, we do not need to calculate $X_{k+1}X_{k+1}^T$, $X_{k+1}P_{k+1}^T$ and $P_{k+1}P_{k+1}^T$.

and P_{k+1} from host to device memory. After the orthogonalization, we orthonormalize the column vectors of W_{k+1} against each other without any HtoD data transfer, and then, we obtain \mathcal{W}_{k+1} by m matrix-vector multiplication operations. So as to calculate the remaining dot products using the \mathcal{W}_{k+1} , we have to transfer X_{k+1} and P_{k+1} from host to device memory. Accordingly, we eliminate the number of the matrices which are transferred from host to device memory by four per iteration. In the following, the method is called ‘Red 1’.

Next, we focus on the operation to orthogonalize the column vectors of W_{k+1} against those of X_{k+1} and P_{k+1} . When we orthogonalize the i -th column vector $\mathbf{w}_{k+1}^{(i)}$, we remove the projection of each column vector of X_{k+1} and P_{k+1} from $\mathbf{w}_{k+1}^{(i)}$ using the result of the dot product. When we operate $\mathbf{w}_{k+1}^{(i)}$ directly, we have to transfer X_{k+1} and P_{k+1} from host to device memory. In order to reduce this data transfer, we represent the operated vector $\mathbf{w}_{k+1}^{(i),*}$ as

$$\mathbf{w}_{k+1}^{(i),*} = \sum_l f_x(l, i) \mathbf{x}_{k+1}^{(l)} + \sum_l f_p(l, i) \mathbf{p}_{k+1}^{(l)} + \sum_l f_w(l, i) \mathbf{w}_{k+1}^{(l)}, \quad (5)$$

and we operate the coefficients instead of calculating the vector $\mathbf{w}_{k+1}^{(i)}$ directly. After the orthogonalization of the column vectors of W_{k+1} against those of X_{k+1} and P_{k+1} , the coefficients are set as $f_x(l, i) = -(\mathbf{x}_{k+1}^{(l)}, \mathbf{w}_{k+1}^{(i)})$, $f_p(l, i) = -(\mathbf{p}_{k+1}^{(l)}, \mathbf{w}_{k+1}^{(i)})$ and $f_w(l, i) = 0(l \neq i), 1(l = i)$. In this operation, when we modify the vector by the operation $\mathbf{t}^* = \mathbf{t} - (\mathbf{t}, \mathbf{s})\mathbf{s}$ for $\|\mathbf{s}\| = 1$, the norm of \mathbf{t}^* is given by $\|\mathbf{t}^*\| = \sqrt{\|\mathbf{t}\|^2 - (\mathbf{t}, \mathbf{s})^2}$. When the norm of a vector becomes smaller than the tolerance in the process of the orthogonalization, we consider the vector to be the combination of other vectors and we eliminate the vector. Afterwards, we execute the orthonormalization of a set of column vectors of W_{k+1} against each other using the MGS method. Here, we can calculate the dot product $z_{i,j} = (\mathbf{w}_{k+1}^{(i),*}, \mathbf{w}_{k+1}^{(j),*})$ by

$$\begin{aligned} z_{i,j} &= \left(\sum_l f_x(l, i) \mathbf{x}_{k+1}^{(l)} + \sum_l f_p(l, i) \mathbf{p}_{k+1}^{(l)} + \sum_l f_w(l, i) \mathbf{w}_{k+1}^{(l)} \right. \\ &\quad \left. \sum_l f_x(l, j) \mathbf{x}_{k+1}^{(l)} + \sum_l f_p(l, j) \mathbf{p}_{k+1}^{(l)} + \sum_l f_w(l, j) \mathbf{w}_{k+1}^{(l)} \right) \\ &= \sum_l f_x(l, i) f_x(l, j) + \sum_l f_p(l, i) f_p(l, j) \\ &\quad + \sum_l \sum_s f_x(l, i) f_w(s, j) (\mathbf{x}_{k+1}^{(l)}, \mathbf{w}_{k+1}^{(s)}) + \sum_l \sum_s f_p(l, i) f_w(s, j) (\mathbf{p}_{k+1}^{(l)}, \mathbf{w}_{k+1}^{(s)}) \\ &\quad + \sum_l \sum_s f_w(l, i) f_x(s, j) (\mathbf{w}_{k+1}^{(l)}, \mathbf{x}_{k+1}^{(s)}) + \sum_l \sum_s f_w(l, i) f_p(s, j) (\mathbf{w}_{k+1}^{(l)}, \mathbf{p}_{k+1}^{(s)}) \\ &\quad + \sum_l \sum_s f_w(l, i) f_w(s, j) (\mathbf{w}_{k+1}^{(l)}, \mathbf{w}_{k+1}^{(s)}). \end{aligned}$$

When $\|\mathbf{w}_{k+1}^{(j),*}\| = 1$, the vectors $\mathbf{w}_{k+1}^{(i),*}$ and $\mathbf{w}_{k+1}^{(j),*}$ become orthogonal by calculating $f_x(l, i) = f_x(l, i) - z_{i,j}f_x(l, j)$, $f_p(l, i) = f_p(l, i) - z_{i,j}f_p(l, j)$ and $f_w(l, i) = f_w(l, i) - z_{i,j}f_w(l, j)$ ($l = 1, 2, \dots, m$). Accordingly, we can orthonormalize all column vectors of W_{k+1} without performing additional dot product operations⁵.

Therefore, the multiplication of the Hamiltonian and an orthonormalized vector $\mathbf{w}_{k+1}^{(i),*}$ can be represented as

$$H\mathbf{w}_{k+1}^{(i),*} = \sum_l f_x(l, i)H\mathbf{x}_{k+1}^{(l)} + \sum_l f_p(l, i)H\mathbf{p}_{k+1}^{(l)} + \sum_l f_w(l, i)H\mathbf{w}_{k+1}^{(l)}. \quad (6)$$

Here, we calculate the remaining dot products $(H\mathbf{x}_{k+1}^{(j)}, \mathbf{w}_{k+1}^{(i),*})$, $(H\mathbf{p}_{k+1}^{(j)}, \mathbf{w}_{k+1}^{(i),*})$ and $(H\mathbf{w}_{k+1}^{(i),*}, \mathbf{w}_{k+1}^{(j),*})$ for constructing the matrix S_A using the coefficients and the results of the dot products in consideration of (5) and (6). Since the dot products except $(H\mathbf{w}_{k+1}^{(i)}, \mathbf{w}_{k+1}^{(j)})$ have already been calculated during the vector update operation, we calculate $(H\mathbf{w}_{k+1}^{(i)}, \mathbf{w}_{k+1}^{(j)})$ using $H\mathbf{w}_{k+1}^{(i)}$ ($= \mathcal{W}_{k+1}^{(i)}$) obtained by executing matrix-vector multiplication. Therefore, we do not need to transfer extra matrix data from host memory for constructing S_A . After we solve the eigenvalue problem for S_A , we transfer four matrices X_{k+1} , \mathcal{X}_{k+1} , P_{k+1} and \mathcal{P}_{k+1} to update X_{k+2} , \mathcal{X}_{k+2} , P_{k+2} and \mathcal{P}_{k+2} . Before the update, we construct W_{k+1} ($= (\mathbf{w}_{k+1}^{(1),*}, \dots, \mathbf{w}_{k+1}^{(m),*})$) and \mathcal{W}_{k+1} ($= (H\mathbf{w}_{k+1}^{(1),*}, \dots, H\mathbf{w}_{k+1}^{(m),*})$) based on (5) and (6) using the transferred matrices and the coefficients f_x , f_p and f_w . As a result, the strategy can reduce HtoD data transfer for four matrices compared to ‘Red 1’. In the following, the method is called ‘Red 2’.

In order to evaluate the effect of the reduction of data transfers, we execute these two methods (‘Red 1’ and ‘Red 2’) and the conventional method (‘Conv’) described in Sect. 2 with the synchronous, the opposite-direction and the asynchronous data transfers under the same condition as in Sect. 2. Here, we set the number of tiles to be 20, and we execute the LOBPCG method with no preconditioner. And then, we show the elapsed time of the orthonormalization, the vector update and the dot product operations in Table 2. In ‘Red1’ and ‘Red2’ methods, some dot product operations are fused with other operations. Therefore, these elapsed times includes the elapsed time of the fused dot product operations. The result demonstrates that the reduction of the number of the data transfer between host and device memory can considerably improve the performance. Moreover, ‘Red2’ method can realize orthonormalization by operating the coefficients of (5) instead of calculating vectors directly, therefore, the method greatly reduce the elapsed time for orthonormalization.

⁵ This orthonormalization operation is equivalent to CholeskyQR method [13].

Table 2. Effect of reduction of data transfers. Here, ‘Sync.’, ‘Opposite’ and ‘Async.’ are represented as the synchronous data transfer, the opposite-directional one and the asynchronous one, respectively.

	Elapsed time (sec)								
	Sync.			Opposite			Async.		
	Conv	Red 1	Red 2	Conv	Red 1	Red 2	Conv	Red 1	Red 2
Orthonormalization +dot product	1.0128	0.5337	0.0008	1.0110	0.5330	0.0008	0.8321	0.4361	0.0008
Vector update +dot product	2.1547	2.5030	2.7597	1.4579	1.8501	2.1254	0.9734	1.4496	1.7005
Remaining dot product	1.0062	0.0158	0.0158	1.0037	0.0156	0.0156	0.7955	0.0158	0.0158
Total	4.1736	3.0525	2.7763	3.4726	2.3988	2.1419	2.6010	1.9015	1.7171

4 Numerical Experiments

In this section, we examine the performance of the LOBPCG method for the Hubbard model on the multi-GPU system in HPE SGI8600 in Japan Atomic Energy Agency. We solve the eigenvalue problems of the Hamiltonian derived from 5×4 -site Hubbard model with 6 up-spins and 6 down-ones. The details of the system are shown in Table 1. Here, the dimension of the Hamiltonian is 1,502,337,600. We attempt to find the eight smallest eigenvalues and the corresponding eigenvectors using a block size of 10 columns. Accordingly, we use the convergence criterion

$$\|H\mathbf{x}_k^{(i)} - \lambda(i)\mathbf{x}_k^{(i)}\| \leq 10^{-6}, \quad i = 1, 2, \dots, 8,$$

where $\lambda(i)$ is an approximate value of the i -th smallest eigenvalues. Here, we set the number of tiles to be 20 and use MPIDirect for communication for the matrix-vector multiplication operation between GPUs. Moreover, we use the zero-shift preconditioner [14, 16]. Since the preconditioner works elementwisely, we utilize ‘Red 2’ as the method for reducing of the data transfers. Table 3 shows the elapsed time of ‘Red2’ method using three types of data transfer. The result indicates that the synchronous data transfer method has the lowest performance of the three methods. The reason is that the conflict for the bus connected between CPU and two GPUs occurs due to performing the data transfer between host and device memory in the same direction simultaneously. The method using the opposite-directional data transfer is performed much faster than the synchronous one, because the method avoids the conflict by the opposite-directional data

Table 3. Parallel performance of LOBPCG method on SGI HPE8600 system. This table shows the total elapsed time, the number of iterations, and the elapsed time per iteration of ‘Red2’ method using the synchronous, the opposite-direction and the asynchronous data transfer operations.

	Elapsed time (sec) (top)		
	Number of iterations (middle)		
	Elapsed time per iteration (sec) (bottom)		
	Synchronous	Opposite	Asynchronous
32 GPU _s	2295.8055	1927.8972	1426.4036
	205	205	205
	11.1991	9.4044	6.9581
64 GPU _s	1233.4289	1067.7207	855.4724
	214	214	214
	5.7637	4.9893	3.9975
128 GPU _s	663.1469	582.9776	503.1989
	210	210	210
	3.1578	2.7761	2.3962
256 GPU _s	371.9728	332.6797	291.2728
	202	202	202
	1.8414	1.6469	1.4419

transfer operations. The method does not overlap the data transfer with the calculation, since its data transfer is a synchronous operation. On the other hand, the asynchronous method can overlap the data transfer with the calculation. Therefore, the method has better performance than the opposite-directional one.

Next, we show the elapsed time of ‘conv’, ‘Red1’ and ‘Red2’ methods using the asynchronous data transfer operation in Table 4. The result indicates that ‘Red2’ is the fastest. And, although Table 2 indicates that ‘Red1’ is more than 10% slower than ‘Red2’, ‘Red1’ is only a few percent slower in this result. The reason is that ‘Red2’ always requires m matrix-vector multiplication operations for calculating $H\mathbf{w}$ by (6), whereas, since ‘Red1’ execute the operations for only independent linearly vectors, there is no need the multiplication for eliminated vectors by the orthonormalization operation.

Table 4. Parallel performance of LOBPCG method on SGI HPE8600 system. This table shows the total elapsed time, the number of iterations, and the elapsed time per iteration of ‘Conv’, ‘Red1’ and ‘Red2’ methods using the asynchronous data transfer operation.

	Elapsed time (sec) (top)		
	Number of iterations (middle)		
	Elapsed time per iteration (sec) (bottom)		
	Conv	Red1	Red2
32 GPU _s	2330.2672	1515.3687	1426.4036
	205	205	205
	11.3672	7.3920	6.9581
64 GPU _s	1314.3161	857.6863	855.4724
	214	214	214
	6.1417	4.0079	3.9975
128 GPU _s	657.6043	523.1674	503.1989
	210	210	210
	3.1314	2.4913	2.3962
256 GPU _s	363.8983	296.9689	291.2728
	202	202	202
	1.8015	1.4701	1.4419

5 Conclusions

we have proposed the parallelization and the tuning strategies of the LOBPCG method, whose almost all operations are performed using GPUs, in order to solve an eigenvalue problem for a large Hamiltonian derived from the Hubbard model using multi-GPU systems. In this research, the dimension of the Hamiltonian is very large and some vectors are stored in host memory. In order to perform the calculations using GPUs in this situation, we have to transfer data between host and device memory as needed. The cost of the data transfer is very large. Therefore, we reduced the transfer operations by considering the algorithm of the LOBPCG method and achieved the improvement of the performance. Moreover, when we execute the conventional method using two processes on each processor on the system as shown in Fig. 5, two processes transfer data in the same direction at the same time. Accordingly, the bus connected between host and device memory is shared with two processes and the throughput per process is limited to about half of the peak throughput. In order to avoid sharing the bus, we have proposed the strategy in which two processes on each processor transfer data in opposite directions. The method has much better performance than the conventional one.

We proposed the strategies in consideration of the property of consuming a small amount of device memory to store Hamiltonian data. Therefore, the pro-

posed strategies can be applied not only to eigenvalue problems for the Hamiltonian derived from the Hubbard model, but also to problems that consume a small amount of device memory to store matrix data or that do not require matrix data to be stored by calculating it per iteration.

In this research, since we mainly focused on the data transfer between host and device memory, all time-consuming operations, that is, matrix-vector multiplications and vector operations, have been executed using GPUs. Recently, the performance of a CPU is improving considerably. Therefore, it is possible to achieve better performance by performing some of calculations using CPUs, especially for problems with a lot of data transfer between host and device memory like the problem in this research. In future work, we plan to investigate the strategy to appropriately distribute the calculations to CPUs and GPUs.

Acknowledgments. This research was partially supported by JSPS KAKENHI Grant Number 18K11345. This research was conducted with the supercomputer HPE SGI8600 in the Japan Atomic Energy Agency.

References

1. Anzt, H., Tomov, S., Dongarra, J.: Accelerating the LOBPCG method on GPUs using a blocked sparse matrix vector product. In: Proceedings of the Symposium on High Performance Computing, pp. 75–82 (2015)
2. Demmel, J., Grigori, L., Hoemmen, M., Langou, J.: Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Sci. Comput.* **34**, A206–A239 (2012). <https://doi.org/10.1137/080731992>
3. Duersch, J.A., Gu, M., Shao, M., Yang, C.: A robust and efficient implementation of LOBPCG. *SIAM J. Sci. Comput.* **40**, C655–C676 (2018). <https://doi.org/10.1137/17M1129830>
4. Furuya, T., Nakatsukasa, Y., Yanagisawa, Y., Yamamoto, Y.: CholeskyQR2: a simple and communication-avoiding algorithm for computing a Tall-Skinny QR factorization on a large-scale parallel system. In: *Scala 2014* (2014)
5. Hetmaniuk, U., Lehoucq, R.: Basis selection in LOBPCG. *J. Comput. Phys.* **228**, 324–332 (2006)
6. Iwata, J.I., et al.: A massively-parallel electronic-structure calculations based on real-space density functional theory. *J. Comput. Phys.* **229**, 2339–2363 (2010). <https://doi.org/10.1016/j.jcp.2009.11.038>
7. Knyazev, A.V.: Preconditioned Eigensolvers - an oxymoron? *Electron. Trans. Numer. Anal.* **7**, 104–123 (1998)
8. Knyazev, A.V.: Toward the optimal Eigensolver: locally optimal block preconditioned conjugate gradient method. *SIAM J. Sci. Comput.* **23**, 517–541 (2001)
9. Montorsi, A. (ed.): *The Hubbard Model: A Collection on Reprints*. World Scientific, Singapore (1992). <https://doi.org/10.1142/1346>
10. Rabbi, F., Daley, C.S., Aktulga, H.M., Wright, N.J.: Evaluation of directive-based GPU programming models on a block Eigensolver with consideration of large sparse matrices. In: Wienke, S., Bhalachandra, S. (eds.) *WACCPD 2019*. LNCS, vol. 12017, pp. 66–88. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-49943-3_4

11. Rasetti, M. (ed.): The Hubbard Model: Recent Results. World Scientific, Singapore (1991). <https://doi.org/10.1142/1377>
12. Siro, T., Harju, A.: Exact diagonalization of the Hubbard model on graphics processing units. *Comp. Phys. Comm.* **183**, 1884–1889 (2012)
13. Stathopoulos, A., Wu, K.: A block orthogonalization procedure with constant synchronization requirements. *SIAM J. Sci. Comput.* **23**, 2165–2182 (2006). <https://doi.org/10.1137/S1064827500370883>
14. Yamada, S., Imamura, T., Machida, M.: 16.447 TFlops and 159-billion-dimensional exact-diagonalization for trapped Fermion-Hubbard model on the earth simulator. In: Proceedings of SC05 (2005)
15. Yamada, S., Imamura, T., Machida, M.: High performance eigenvalue solver in exact-diagonalization method for Hubbard model on CUDA GPU. In: Joubert, G.R., Leather, H., Parsons, M., Peters, F., Sawyer, M. (eds.) *Parallel Computing: On the road to Exascale. Advances in Parallel Computing*, vol. 27, pp. 361–369. IOS (2016). <https://doi.org/10.3233/978-1-61499-621-7-361>
16. Yamada, S., Imamura, T., Machida, M.: Communication avoiding Neumann expansion preconditioner for LOBPCG method: convergence property of exact diagonalization method for Hubbard model. In: Bassini, S., Danelutto, M., Dazzi, P., Joubert, G.R., Peters, F. (eds.) *Parallel Computing is Everywhere. Advances in Parallel Computing*, vol. 32, pp. 27–36. IOS (2018). <https://doi.org/10.3233/978-1-61499-843-3-27>
17. Yamada, S., Imamura, T., Machida, M.: High performance eigenvalue solver for Hubbard model: tuning strategies for LOBPCG method on CUDA GPU. In: Foster, I., Joubert, G.R., Kučera, L., Nagel, W.E., Peters, F. (eds.) *Parallel Computing: Technology Trends. Advances in Parallel Computing*, vol. 36, pp. 105–113. IOS (2020). <https://doi.org/10.3233/APC200030>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Vapor Condensation Under Electric Field: A Study Using Molecular Dynamics Simulation

Pengyu Wang^(✉) and Zhong Chen

School of Materials Science and Engineering, Nanyang Technological University,
50 Nanyang Avenue, Singapore 639798, Singapore
n2007034k@e.ntu.edu.sg

Abstract. The condensation of water vapor on the substrate surface under electric field is studied by molecular dynamics simulation, and a series of behaviors of water molecules during condensation were studied, such as nucleation, growth and coalescence. In the process of condensation, there will be some small clusters, whose size increases with the increase of time, and under the action of the movement of water molecules in vapor, the clusters move irregularly on the substrate surface and coalesced into larger clusters. And the droplets will be stretched along the direction of the electric field. Interestingly, the condensation will decrease with the increase of the electric field strength under the electric field perpendicular to the surface. The results also show that the orientations of water molecule dipole are closely related to the direction of electric field, indicating that the electric field causes the realignment of water molecules. The research shows that the electric intensity will have great impact on vapor condensation, which provides guidance for reversible adjustment of vapor condensation and the design of intelligent surface.

Keywords: Condensation · Molecular dynamics simulation · Electric field · Cluster

1 Introduction

Vapor condensation is closely related to our daily life and can be utilized for water collection [1], thermal management [2], water desalination [3] and so on. Therefore, mechanism of water vapor condensation is very important for the rational utilization of condensation, and the efficient condensation has also attracted the interest of many researchers.

A large number of studies show that the roughness and chemical properties of the condensation surface have important impact on the condensation efficiency [4–9]. The development of molecular dynamics simulation provides a new method for the study of water molecular condensation. For example, Xu et al. studied the condensation of water vapor on the V-shaped groove by molecular dynamics simulation [4], the results show that the wetting modes of clusters are determined by the intrinsic contact angle and the cross sectional angle of the surface. Another study by the same team shows that the

formation and growth of clusters in the process of water vapor condensation increase with the increase of substrate hydrophilicity [5]. Gao et al. explored the condensation of water vapor on the nanostructured surfaces with hybrid wettability areas and found that the nanostructured surfaces with hybrid wettability areas has better heat transfer performance [6]. Huang et al. discussed the effects of pillar height, spacing and substrate wettability on argon condensation [7]. However, the influence of external field is ubiquitous and will have a great impact on the condensation of water vapor, such as electric, temperature, force.

At present, studies exist mainly on the influence of external electric field on the static and dynamic behavior of sessile droplets on the surface [10–15] and water evaporation [16]. The research of Yan et al. [10] shows that the voltage amplitude and frequency of tangential AC electric field are important factors affecting the dewetting of droplets. The dynamics behaviors of droplets on flexible graphene sheets under constant and alternating electric fields with different amplitudes and frequencies are studied by Kargar [11] et al., and the results show that droplets elongated in the direction of electric field. Wang et al. [12] simulated the condensation process of water molecules under the action of an electric field perpendicular to the surface by molecular dynamics simulation and found that the condensation could be inhibited by vertical electric field.

Water molecules are easy to be affected by electric field because of the polarity, and electric field is an easily available clean energy, which provides a premise for its large-scale utilization, such as electrostatic spray, electrohydrodynamic atomization. The existence of external field also provides a new method to dynamically regulate the condensation of water molecules. However, the mechanism of the vapor condensation under external field is still not well understood. In this paper, the condensation process of water molecules under constant electric field perpendicular to the surface is studied by molecular dynamics simulation. The growth of maximum cluster size, number of clusters, dipole moment and the temperature of water molecules are calculated to quantify the condensation process. The research results are also useful for the efficient utilization of heat and mass transfer.

2 Model and Methods

In this paper, the molecular dynamics simulation software LAMMPS is used to simulate the condensation process of water vapor under the action of electric field [17, 18]. The structure of the simulation system is shown in Fig. 1. The simulation system consists of three parts, including an upper substrate (hot surface), a lower substrate (cold surface) and water molecules. In order to improve the calculation efficiency, the structure of copper atoms is used to establish the models of the cold surface and hot surface, in which the lattice is 3.615 Å, the thickness of the cold surface and hot surface is 10.845 Å, and the lateral dimension is 253 * 253 Å, consisting of 58800 atoms, [6, 19, 20]. Meanwhile, to improve the condensation rate of water molecules, 10000 water molecules are placed in the simulation systems.

The simple point charge/extension (SPC/E) model, which has been validated in previous studies, is chosen to describe the interaction between the water and water [21–23]. The calculation method of the interaction between water molecules in the model

is consistent with our previous research [21, 24]. The interaction between substrate atoms and water molecules is calculated by Lennard-Jones (L-J) potential, where the energy coefficient (ϵ) is 0.2 kJ/mol, and the distance coefficient (σ) is 2.891 Å. The interaction between the substrate atoms is also described by the L-J potential, in which $\epsilon_{S-S} = 4.72$ kJ/mol, $\sigma_{S-S} = 2.616$ Å [25], and there is no interaction between the upper substrate atoms and lower substrate atoms. In the simulation, the cutoff distances are set as 12 Å.

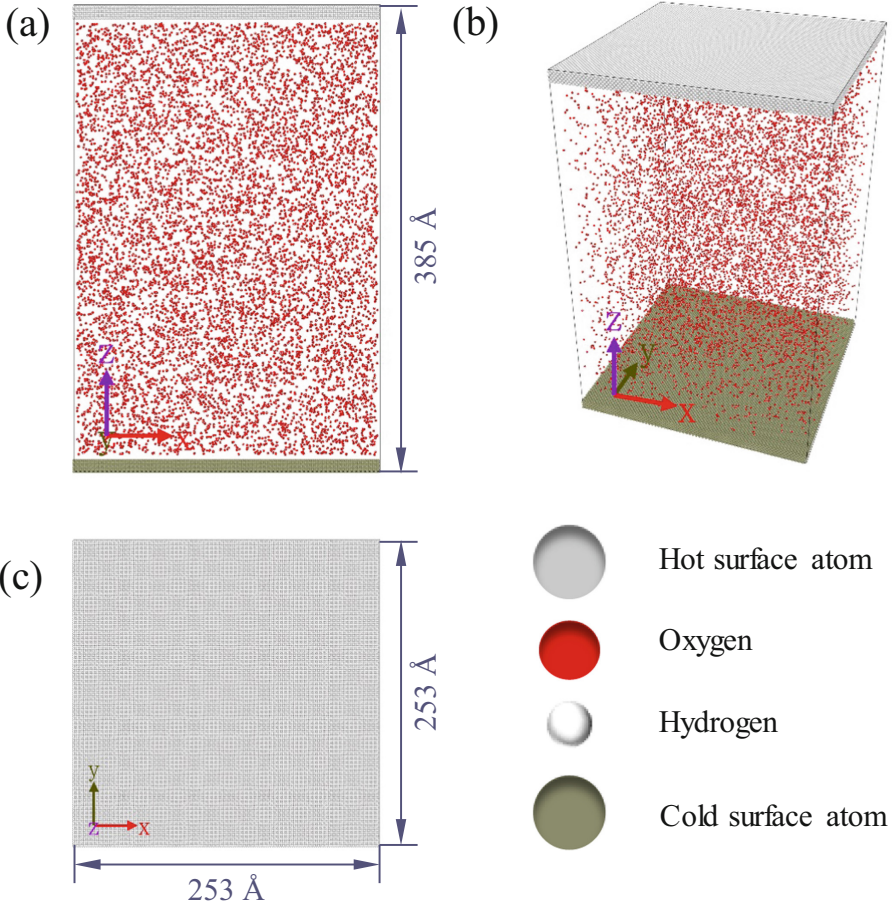


Fig. 1. Schematic diagram for the simulation system. (a) Front view. (b) Perspective view. (c) Top view.

In the process of simulation, the periodic boundary conditions are applied in all directions. Meanwhile, the velocity-Verlet algorithm is used to solve the Newtonian motion equation with a time step of 1 femtosecond (fs). The particle-particle particle-mesh (PPPM) method is used to calculate the long-range electrostatic interactions with

an accuracy up to 10⁻⁴. And to improve the calculation efficiency, SHAKE algorithm is used to fix the bonds and angles of water molecules.

And all the simulations are performed in two stages: First, the equilibration phase is carried out for a total of 0.2 ns (ns) in a NVT ensemble, and the temperature of water molecules and substrate atoms are controlled at 500 K. Then the thermostat applied on the water vapor is removed, the temperatures of the lower and upper surfaces in the system are controlled at 300K and 500K, respectively. In order to observe the condensation of water molecules faster, the temperature difference of the substrates is set larger. At the same time, the electric field is applied to the system. In our study, constant electric field perpendicular to the surface is applied to study the effect of electric field on water vapor condensation. And the simulation time for condensation is extended to 4 ns to see the condensation process clearly.

The growth of clusters is an important index to study the condensation process. In this study, the Stillinger criterion is used to define clusters, two water molecules are determined to be located in one cluster when the distance between oxygen atoms in two water molecules is 3.36 Å [26].

3 Results and Discussion

In this study, the condensation process of water vapor is simulated under the action of vertical electric field, and some snapshots of the simulation systems are selected to show the condensation process. Only part of the simulation system is shown in Fig. 2 because of the large size of the system. Figure 2 shows the condensation process of water vapor in different electric fields, and the direction of the electric field points to the negative direction of the y-axis. Water molecules move irregularly in the simulation process. When water molecules impact the cold surface, part of the energy of water molecules is converted into heat energy, resulting in the reduction of the speed of water molecules, which may be adsorbed on the cold surface or rebound back to the vapor. At this time, the motions of water molecules are influenced by electric field force, van der Waals force between substrate atoms and water molecules, and the interaction between water molecules. Due to the temperature difference between the cold surface and the hot surface, water molecules tend to adsorb on the cold surface and gradually form clusters on the cold surface. With the increase of electric field intensity, the condensation of water molecules is affected.

It can be seen from Fig. 2 that the clusters formed by condensation will be gradually elongated with the increase of electric field intensity. For example, the stretching of clusters is not obvious when the electric field intensity is 0.4 V/nm. When the electric field intensity is 0.8 V/nm, the deformation of droplets is obvious and becomes cylindrical. When the electric field intensity is 1.0 V/nm, the condensed clusters will leave the substrate surface, indicating a great influence of the electric field.

In the subsequent simulation, the condensation of water vapor in the electric field pointing to the positive direction of Y axis is also simulated. The results show that the electric field pointing to the positive direction of Y axis also inhibits the condensation of water vapor, and the effect is more obvious than that in the negative direction. It stretches the water droplets stretch along the positive direction of the Y axis. Our findings are

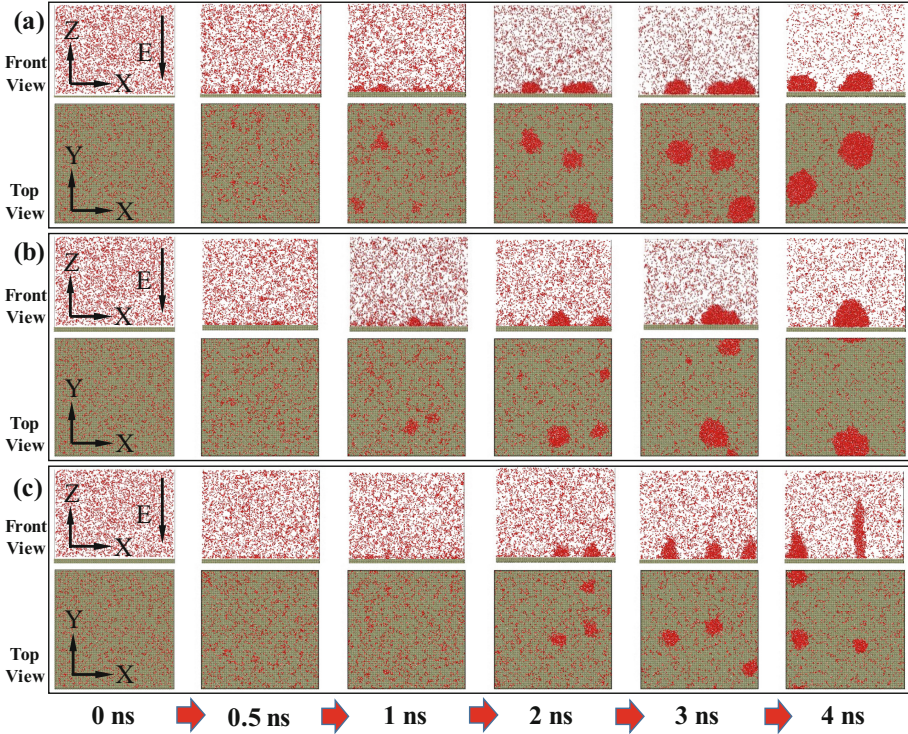


Fig. 2. Snapshots (Front view and Top view) of the condensation process with different constant electric fields. (a) $E = 0.0$ V/nm. (b) $E = 0.4$ V/nm. (c) $E = 0.8$ V/nm.

consistent with previous studies on sessile droplets, in which the vertical electric field in the positive and negative directions was found to stretch and deform the droplets, and the deformation is more obvious under the electric field in the positive direction [27–29]. At the same time, it is obvious from Fig. 2 that the condensation speed of water molecules decreases gradually with the increase of electric field intensity. When the electric field intensity is small, the time that clusters start to form is not much different from that without electric field. However, clusters appear much later when the electric field strength increases to 0.8 V/nm. Our results show that the electric field inhibits the condensation of water vapor.

To more specifically quantify the impact of electric field on water molecular condensation, the growth of the maximum cluster size and the number of clusters in the system are calculated in the condensation process, and the results are shown in Fig. 3. Figure 3 (a) shows the temporal evolution of maximum cluster size under constant electric fields. The growth curve of the maximum cluster size increases linearly in some stages, indicating that the number of water molecules in the system is sufficient to simulate accurately the condensation process, and the abrupt increase in the curve indicates that adjacent clusters have merged. Figure 3 (a) shows the merging process of adjacent clusters when the electric field intensity is 0.0 V/nm. This also shows that the growth of the maximum

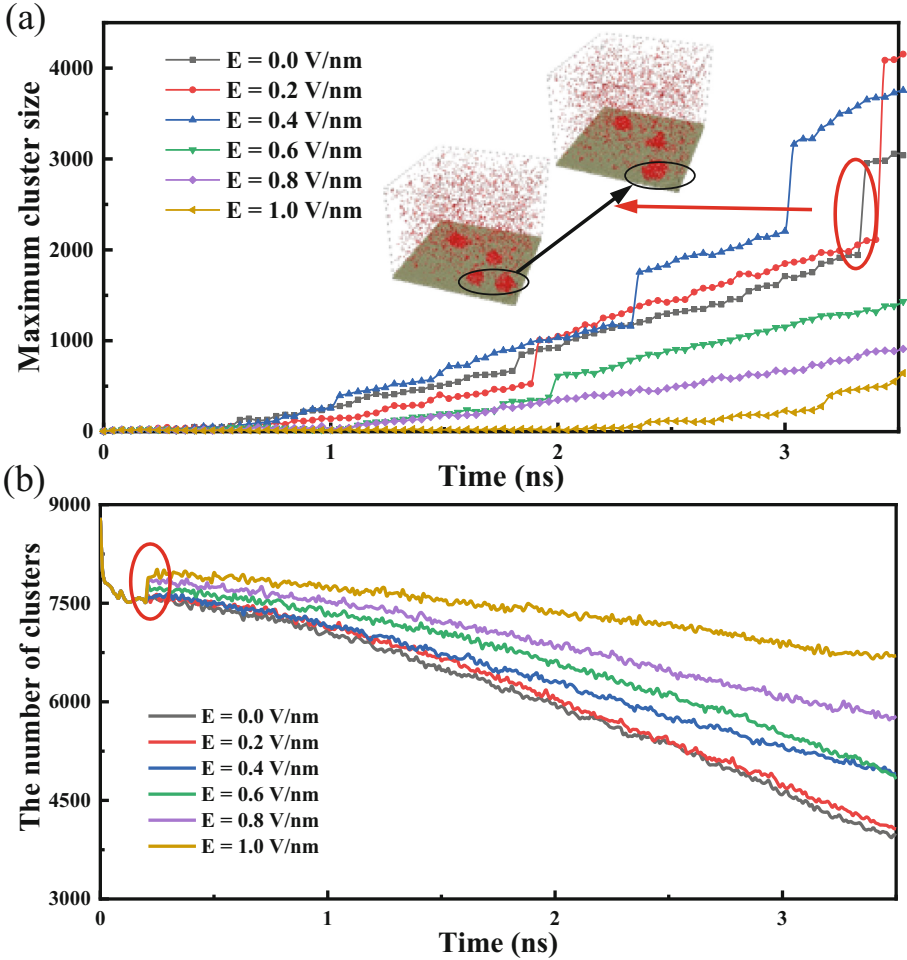


Fig. 3. (a) Temporal evolution of maximum cluster size under constant electric fields, the inset shows the coalescence process of clusters ($E = 0.0$ V/nm). (b) Temporal evolution of the number of clusters under constant electric fields.

cluster size is more obvious with the increase of electric field, except for the electric field intensity is 0.2 and 0.4 V/nm. From the condensation process, it is found that the reason for the abnormal maximum cluster size is that there is only one large cluster in the system when electric field intensities are 0.2 and 0.4 V/nm. While for the simulation when the electric field intensity is 0 V/nm, there are two large clusters on the cold surface in the condensation process. Figure 3 (b) shows the temporal evolution of the number of clusters under constant electric fields. The red circle indicates that the increase of electric field will destroy the clusters in the system, resulting in a sudden increase of the number of clusters. It is clear that the greater the electric field intensity, the greater the increase of the curves in the red circle. The results imply that the formation of clusters

is inhibited with the increase of electric field intensity. This result corresponds well to the analysis in Fig. 2.

The polarity characteristics of water molecules show that water molecules are easy to be affected by electric field. Therefore, the dipole moments of water molecules are also calculated to quantify the effect of electric field on water molecules and to explain the growth and deformation of clusters. Figure 4 shows the average of the $\cos(\theta)$ values of the water molecules under constant electric field (θ is the angle between the dipole moment of water molecule and the direction of electric field.). When the electric field intensity is 0.0 V/nm, the average of the $\cos(\theta)$ is close to 0.0, indicating that water molecules are in a disordered state. With the increase of the electric field intensity, the average value of $\cos(\theta)$ begins to increase gradually, indicating that the structure of water molecules rearranges under the action of the electric field, and the dipole moment of water molecules begin to align the direction of the electric field. At this point, the hydrogen atoms in water molecules point to the direction of the electric field, while the direction of oxygen atoms is opposite to the direction of the electric field. This characteristic makes the clusters stretch and grow along the direction of the electric field.

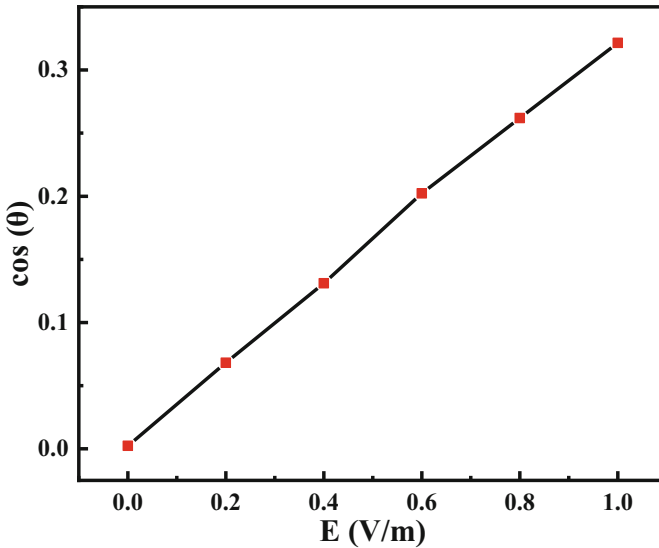


Fig. 4. The average of the $\cos(\theta)$ values of the water molecules under constant electric field.

Next, we collected the temperature of each part of the system in the simulation process as shown in Fig. 5 (a), which displays the temporal evolution of water molecule temperature, and the cold and hot surfaces under different electric field intensities. For the system with electric field intensity of 0.0 V/nm, the temperature of water molecules begins to decrease from 500K with the collision between water molecules and cold surface when the temperature of the cold surface is controlled at 300K. When the electric field is applied, the temperature of water molecules rises instantaneously, indicating that water molecules acquire energy to raise the water temperature, as shown in the red circle

in Fig. 5(a). This corresponds to the sudden increase in the number of clusters in Fig. 3 (b). The temperature of water molecules begins to decrease gradually as the simulation progresses. Figure 5(a) demonstrates that the higher the electric field strength, the higher the water molecule temperature during the whole simulation process. Obviously, this is not conducive to the condensation of water molecules. Interestingly, the slope of the temperature curve corresponding to the simulation system with electric field intensity of 1.0 V/nm becomes very small after 3 ns. This is because the electric field force plays a dominant role, the clusters are stretched and leave the cold surface, as shown in Fig. 5 (b) and (c), which is not conducive to heat transfer. The literature also draws similar conclusions from the released accumulated energy [12].

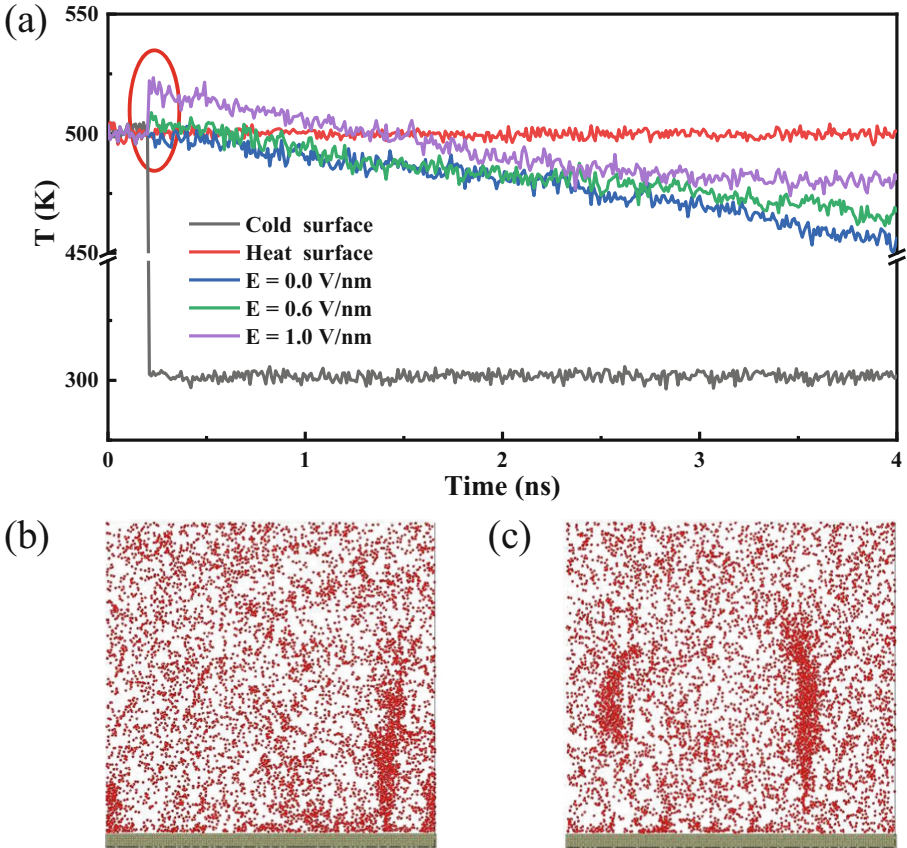


Fig. 5. (a) Temporal evolution of the temperature (T) of water molecules in simulations. (b) The snapshot of the simulation system under the electric field of 1.0 V/nm (Front view), $T = 3.5$ ns. (c) The snapshot of the simulation system under the electric field of 1.0 V/nm (Front view), $T = 4$ ns.

Then, the mean square displacement of water molecules in the Z direction (MSD_z) is also calculated to evaluate the effect of electric field on condensation, as shown in

Fig. 6. The simulation process in the Fig. 6 is divided into two parts: the first part is the equilibrium process of condensation system, and then the water vapor begins to condense under electric field. The results show that under the action of electric field, the MSDz curve of water molecules in the Z direction gradually decreases with the increase of electric field intensity, which shows that the diffusion of water molecules in the Z direction is restrained and the probability of droplet collision with the substrate is reduced. This is because the strong electric field induces water molecules to form an ordered structure pointing to the electric field direction, which is not conducive to the formation of clusters, At the same time, this reveals why the condensation efficiency decreases with the increase of electric field intensities.

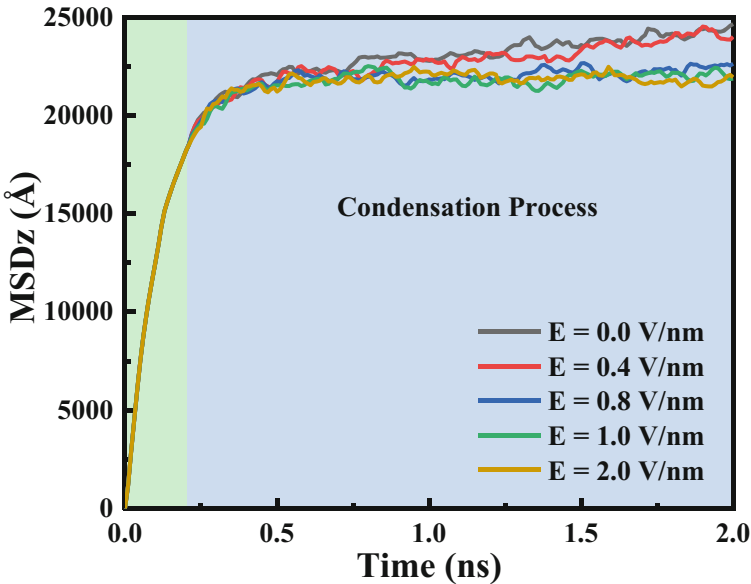


Fig. 6. The mean square displacement of water molecules in the Z direction (MSDz) under constant electric field.

4 Conclusion

In this study, the condensation process of water vapor under electric field is studied by molecular dynamics simulation. The influence of electric field on water vapor condensation is quantitatively evaluated by calculating the growth of the maximum cluster size, the number of clusters, the dipole moment of water molecules and the system temperature. The results show that the existence of vertical electric field promotes the rearrangement of water molecules along the direction of electric field, which makes the clusters grow along the direction of electric field. The formed clusters are stretched, which may even lead to the separation of clusters from the substrate surface. At the same time, the existence of vertical electric field increases the temperature of water molecules and inhibits

the condensation of water vapor. And the condensation efficiency of droplets decreases with the increase of electric field intensity.

Acknowledgement. The computational work for this article was partially performed on resources of the National Supercomputing Centre, Singapore (<https://www.nscg.sg>).

References

1. Ju, J., Bai, H., Zheng, Y., Zhao, T., Fang, R., Jiang, L.: A multi-structural and multi-functional integrated fog collection system in cactus. *Nat. Commun.* **3**(1), 1247 (2012)
2. Miljkovic, N., Preston, D.J., Enright, R., Wang, E.N.: Electric-field-enhanced condensation on superhydrophobic nanostructured surfaces. *ACS Nano* **7**(12), 11043–11054 (2013)
3. Khawaji, A.D., Kutubkhanah, I.K., Wie, J.M.: Advances in seawater desalination technologies. *Desalination* **221**(1), 47–69 (2008)
4. Xu, W., Lan, Z., Peng, B.L., Wen, R.F., Ma, X.H.: Effect of nano structures on the nucleus wetting modes during water vapour condensation: from individual groove to nano-array surface. *RSC Adv.* **6**(10), 7923–7932 (2016)
5. Xu, W., Lan, Z., Peng, B.L., Wen, R.F., Ma, X.H.: Effect of surface free energies on the heterogeneous nucleation of water droplet: a molecular dynamics simulation approach. *J. Chem. Phys.* **142**(5), 054701 (2015)
6. Gao, S., Liu, W., Liu, Z.: Tuning nanostructured surfaces with hybrid wettability areas to enhance condensation. *Nanoscale* **11**, 459–466 (2019)
7. Huang, D., Quan, X., Cheng, P.: An investigation on vapor condensation on nanopillar array surfaces by molecular dynamics simulation. *Int. Commun. Heat Mass Transfer* **98**, 232–238 (2018)
8. Starostin, A., Valtsifer, V., Barkay, Z., Legchenkova, I., Danchuk, V., Bormashenko, E.: Drop-wise and film-wise water condensation processes occurring on metallic micro-scaled surfaces. *Appl. Surf. Sci.* **444**, 604–609 (2018)
9. Ranathunga, D.T.S., Shamir, A., Dai, X., Nielsen, S.O.: Molecular dynamics simulations of water condensation on surfaces with tunable wettability. *Langmuir* **36**(26), 7383–7391 (2020)
10. Yan, X., Li, J., Li, L., Huang, Z., Wang, F., Wei, Y.: Droplet condensation on superhydrophobic surfaces with enhanced dewetting under a tangential AC electric field. *Appl. Phys. Lett.* **109**(16), 161601 (2016)
11. Kargar, M., Lohrasebi, A.: Deformation of water nano-droplets on graphene under the influence of constant and alternative electric fields. *Phys. Chem. Chem. Phys.* **19**, 26833–26838 (2017)
12. Wang, Q., Xie, H., Hu, Z., Liu, C.: The impact of the electric field on surface condensation of water vapor: insight from molecular dynamics simulation. *Nanomaterials* **9**(1), 64 (2019)
13. Ren, H., Zhang, L., Li, X., Li, Y., Wu, W., Li, H.: Interfacial structure and wetting properties of water droplets on graphene under a static electric field. *Phys. Chem. Chem. Phys.* **17**, 23460–23467 (2015)
14. Zhang, B., Wang, S., He, X., Yang, Y., Wang, X.: Dynamic spreading of a water nanodroplet on a nanostructured surface in the presence of an electric field. *J. Mol. Liq.* **333**, 116039 (2021)
15. Yuan, Q., Zhao, Y.: Precursor film in dynamic wetting, electrowetting, and electro-elastocapillarity. *Phys. Rev. Lett.* **104**(24), 246101 (2010)

16. Hens, A., Biswas, G., De, S.: Evaporation of water droplets on Pt-surface in presence of external electric field—a molecular dynamics study. *J. Chem. Phys.* **143**(9), 094702 (2015)
17. Plimpton, S.: Fast parallel algorithms for short-range molecular dynamics. *J. Comput. Phys.* **117**(1), 1–19 (1995)
18. LAMMPS Molecular Dynamics Simulator. <http://lammps.sandia>. Accessed 12 Dec 2018
19. Wang, P., He, L., Sun, X., Lv, H., Wang, Z.: Influence of trapezoidal cavity on the wettability of hydrophobic surface: a molecular dynamics study. *Langmuir* **37**(12), 3575–3584 (2021)
20. Zhu, C., et al.: Controlling states of water droplets on nanostructured surfaces by design. *Nanoscale* **9**, 18240–18245 (2017)
21. Wang, P., Sun, X., Lv, H., Ma, S., Wang, Z.: Investigation of surface wettability and their influencing mechanisms under vibration field: a molecular dynamics simulation study. *Comput. Mater. Sci.* **197**, 110615 (2021)
22. Berendsen, H.J.C., Grigera, J.R., Straatsma, T.P.: The missing term in effective pair potentials. *J. Phys. Chem.* **91**(24), 6269–6271 (1987)
23. Zhao, Y., Yuan, Q.: Statics and dynamics of electrowetting on pillar-arrayed surfaces at the nanoscale. *Nanoscale* **7**(6), 2561–2567 (2015)
24. Wang, P., He, L., Wang, Z.: The effect of surface structure and arrangement on wettability of substrate surface. *Colloids Surf. A* **614**, 126165 (2021)
25. Heinz, H., Vaia, R.A., Farmer, B.L., Naik, R.R.: Accurate simulation of surfaces and interfaces of face-centered cubic metals using 12–6 and 9–6 Lennard-Jones potentials. *J. Phys. Chem. C* **112**(44), 17281–17290 (2008)
26. Stillinger, F.H.: Rigorous basis of the Frenkel-Band theory of association equilibrium. *J. Chem. Phys.* **38**(7), 1486–1494 (1963)
27. Song, F., Ma, L., Fan, J., Chen, Q., Lei, G., Li, B.Q.: Electro-wetting of a nanoscale water droplet on a polar solid surface in electric fields. *Phys. Chem. Chem. Phys.* **20**(17), 11987–11993 (2018)
28. Zhang, Z., Dong, X., Ye, H., Cheng, G., Ding, J., Ling, Z.: Wetting and motion behaviors of water droplet on graphene under thermal-electric coupling field. *J. Appl. Phys.* **117**(7), 074304 (2015)
29. Zong, D., Yang, Z., Duan, Y.: Wettability of a nano-droplet in an electric field: a molecular dynamics study. *Appl. Therm. Eng.* **122**, 71–79 (2017)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





The Effect of Wing Mass and Wing Elevation Motion During Insect Forward Flight

Jie Yao^(✉)  and K. S. Yeo

Department of Mechanical Engineering, National University of Singapore, Singapore, Singapore
mpeyaoj@nus.edu.sg

Abstract. This paper is concerned with the numerical simulation of the forward flight of a high Reynolds number flapping-wing flyer, modelled after the hummingbird hawkmoth (*Macroglossum stellatarum*). The numerical model integrated a Navier-Stokes solver with the Newtonian free-body dynamics of the model insect. The primary cyclic kinematics of wings were assumed to be sinusoidal for simplicity here, which comprises sweeping, elevating and twisting related wing actions. The free flight simulation is very computationally intensive due to the large mesh scale and the iterative solution for the FSI problem, so parallelization is essential in the numerical simulation. Two parallelization techniques are used in current simulation, i.e., open multi-processing (OpenMP) and graphics processing units (GPU) acceleration. The forward flight mainly consists of two stages, i.e., the body pitching down from the normal hovering posture and the following forward acceleration. During this process, the effect of the wing mass and the wing elevation motion is very important, which is investigated in detail. It is found that Oval-shaped wing elevating motion can help to generate large pitching down moment so that the flyer can quickly adjust its orientation for forward acceleration. Moreover, wing mass tends to magnify the effect and prohibits the growth of pitching down velocity, which is favourable aspect. The present study provides detailed information of the coupled dynamics of fluid and flyer in free flight condition, as well as offers a prospective approach that could complement existing experiments in a wider study of insect flight and maneuver.

Keywords: Insect free flight · CFD simulation · Parallel computation · OpenMP · GPU acceleration

1 Introduction

Over the past few decades, concerted efforts have been made by researchers to unravel the physical phenomena underlying the insects' fascinating flight performances. Flapping-wing flight appears to be more advantageous in terms of its maneuverability and efficiency compared to conventional fixed-wing flight, especially for small size flyers. It is believed that successful application of insects' aerodynamics could revolutionize the design of Micro Air Vehicle (MAV).

The earliest study of flapping wing aerodynamics attempted to estimate the forces generated by the wings using quasi-steady aerodynamics of conventional fixed-wing flyers [1–5]. Experimental biomechanics and fluid dynamics have played a key role in the

study of flapping flight since the 1920s. The experimental studies by Willmott & Ellington [6, 7], Dickinson et al. [8], Fry et al. [9] and others have measured the wing kinematics and explored the unsteady aerodynamics related to flapping wing flight in great details. Computational Fluid Dynamics (CFD) arguably offers a complimentary and sometimes more expedient approach to access the unsteady aerodynamics at the scales of the insects. Benefiting from the continuously improving computing hardware and technology, the development of CFD has advanced rapidly in the past several decades. Full-fidelity CFD-FSI (computational fluid dynamics with fluid structure/body interaction) simulation is one promising approach to study the free flight of a flapping-wing insect, where the interactions between flyer and flow can be well resolved in a coupled manner. Wu et al. [10] successfully simulated the controlled free hovering flight of a model fruit fly with all six degrees of freedom, and investigated its ‘real-time’ aerodynamics and the role of active feedback control. Yao and Yeo [11, 12] subsequently extended the hovering model of Wu et al. to study the free longitudinal flights and simple manoeuvres (saccadic yawing and rapid sideslipping) of the fruit fly. Yao and Yeo [13, 14] adopted a similar numerical framework to study the hovering and the forward flight and sideslip manoeuvre of a hummingbird hawkmoth, wherein the significant reciprocating wing mass of the hawkmoth was also accounted for.

In this paper, we shall be concerned with simulating the forward flight of a model hummingbird hawkmoth, *Macroglossum stellatarum*, where the effect of wing mass and wing elevation motion is studied. The simulation of insects in free flight remains highly challenging due to their dynamical complexities and high computational cost. To accelerate the computation, two parallelization techniques are used in current simulation, i.e., open multi-processing (OpenMP) and graphics processing units (GPU) acceleration.

Section 2 gives the morphological details of the model hawkmoth and the basic methodology for the numerical simulation. Section 3 presents the results for the forward flight of the model hawkmoth. The key conclusions from present study are summarized in Sect. 4.

2 Methodology

2.1 Governing Equations and Numerical Discretization

The hummingbird hawkmoth (*Macroglossum stellatarum*) is adopted as the model flapping wing flyer in the present work. It is a good representative of the family *Sphingidae*, which usually exhibits superior flight performance among flying insects with relatively high Reynolds number ($Re \approx 3000$). The flapping wing motion of hummingbird hawkmoth has a frequency (f) of approximately 70 Hz and a stroke amplitude (Φ) of about 115° according to available literature [15]. The model insect has a wing length (L) of 20.2 mm and total mass (M) of 0.21g, where the wings’ mass takes about 4.66%.

Wing motion is specified by three flapping angles: the stroke/sweep angle (ϕ), the elevation angle (θ) and the twist angle (ψ). The wing angles ϕ , θ , ψ are basically the Euler angles. They specify the action of the wings during flight and are thus prescribed functions or actively-controlled function of time t . Low-order Fourier series comprising simple sinusoidal functions are used to describe the wing kinematics of the present model

flyer. The wing angle functions may thus be represented as follows:

$$\begin{cases} \phi(t) = -\frac{\Phi}{2} \cos(2\pi t) \\ \theta(t) = \theta_0 + \theta_{a1} \cos(2\pi t) + \theta_{b1} \sin(2\pi t) + \theta_{a2} \cos(4\pi t) + \theta_{b2} \sin(4\pi t) \\ \psi(t) = \frac{\pi}{2} - \frac{\pi}{4} \sin(2\pi t) \end{cases} \quad (1a-c)$$

where t is the non-dimensional time. The θ_a and θ_b are parameters that define the shape of wing tip trajectory, such as U-shaped, oval-shaped and ∞ -shaped paths.

The fluid flow around the model insect is governed by the three-dimensional incompressible Navier-Stokes equations, in the non-dimensional Arbitrary Lagrangian-Eulerian (ALE) form given here:

$$\begin{cases} \nabla \cdot \mathbf{u} = 0 \\ \partial_t \mathbf{u} = -(\mathbf{u} - \mathbf{u}^g) \cdot \nabla \mathbf{u} + \frac{1}{\text{Re}} \nabla^2 \mathbf{u} - \nabla p \end{cases} \quad (2a-b)$$

where \mathbf{u} and p represent the velocity and pressure field of the fluid domain respectively, and \mathbf{u}^g is the convection velocity of the computational node. A projection-based method is adopted to solve above equations, where the insect body and wings are modelled as non-slip surfaces $\Gamma(t)$, on which $\mathbf{u}(\mathbf{x}) = \mathbf{u}^g(\mathbf{x})$. The pressure boundary condition is given by $\mathbf{n} \cdot \nabla p = -\mathbf{n} \cdot \mathbf{a}$, where \mathbf{a} is the acceleration of the surface node. Neumann-type boundary conditions are applied at the far field boundaries of the flow domain.

The free flight of the model insect is governed by Newtonian dynamics. Assuming the body and wings to be essentially rigid, and the wings to have negligible mass, the kinematic and dynamic equations for flight are given by:

$$\begin{cases} \frac{d}{dt} \mathbf{X}_B(t) = \mathbf{V}_B(t) \\ \frac{d}{dt} \Theta(t) = [\mathbf{K}(\Theta)] \cdot \omega(t) \\ \frac{d}{dt} (M \cdot \mathbf{V}_B) = -Mg\mathbf{k}_g + \mathbf{F}_A \\ \frac{d}{dt} \left(\overset{\leftrightarrow}{\mathbf{I}}^B \cdot \omega \right) = \mathbf{T}_A^B \end{cases} \quad (3a-d)$$

where $\mathbf{X}_B(t)$ is the position of body centre and also the centre of mass (CoM) at time t ; $\Theta(t)$ the orientation vector of the flyer and $[\mathbf{K}(\Theta)]$ the transformation matrix relating the rate of change of $\Theta(t)$ to the angular velocity $\omega(t)$ of the body. The $\overset{\leftrightarrow}{\mathbf{I}}^B$ and $\overset{\leftrightarrow}{\mathbf{I}}^B \cdot \omega$ denote the inertia tensor and the angular momentum of the flyer about its CoM. The $\mathbf{F}_A(t)$ and $\mathbf{T}_A^B(t)$ denote the net resultant aerodynamic force and moment respectively acting on the flyer about CoM, which are obtained from the flow field solver. The closed form expressions for the dynamics with wing mass are rendered highly complex because of the complex kinematics of the two wings, plus the relative shifting of the insect's CoM within the body frame. The reader is referred to [16] for its derivation and further details.

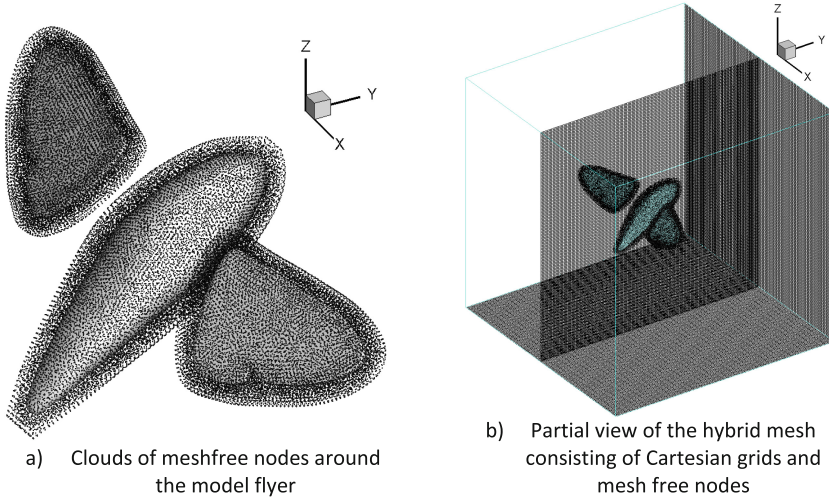


Fig. 1. Grid system for numerical simulation.

The configuration of the mesh used in current simulation is shown in Fig. 1. A uniform Cartesian background grid had been adopted for the domain – Fig. 1b gives a partial view to highlight the placement of the model fly. The whole mesh system consists of 401^3 Cartesian background grids and around 60000 moving nodes near the model insect.

The complex fluid-structure interaction (FSI) implies the need to iterate the solution process between flow solver $\mathbf{F}[\Gamma(t)]$ and the dynamic solver $\mathbf{S}\{\mathbf{F}[\Gamma]\}$ of Eq. (3) at each time step to determine the updated/new configuration of the flyer $\Gamma(t)$. The iteration is carried at each time step until the change between consecutive estimates of Γ is smaller than a given tolerance.

2.2 Projection Method

Projection method is applied to solve Eq. (2), which is further decomposed into Eq. (4a) and Eq. (4b). The pressure Poisson Eq. (4c) is then obtained by taking divergence of Eq. (4b) and invoking the continuity equation.

$$\left\{ \begin{array}{l} \frac{\mathbf{u}^* - \mathbf{u}^n}{\Delta t} = \frac{1}{2} \left\{ \begin{array}{l} \left[-(\mathbf{u} - \mathbf{u}^g) \cdot \nabla \mathbf{u} + \frac{1}{\text{Re}} \nabla^2 \mathbf{u} \right]^{n+1} \\ + \left[-(\mathbf{u} - \mathbf{u}^g) \cdot \nabla \mathbf{u} + \frac{1}{\text{Re}} \nabla^2 \mathbf{u} - \nabla p \right]^n \end{array} \right\} \\ \frac{\mathbf{u}^{n+1} - \mathbf{u}^*}{\Delta t} = -\frac{1}{2} \nabla p^{n+1} \\ \nabla^2 p^{n+1} = \frac{2}{\Delta t} \nabla \cdot \mathbf{u}^* \end{array} \right. \quad (4a-c)$$

Important steps of the implementation are summarized below.

STEP 1. Compute intermediate velocity field \mathbf{u}^*

For interior nodes: solve Eq. (4a).

For solid boundary nodes: $\mathbf{u}^* = \mathbf{u}^{n+1} = \mathbf{u}^{g,n+1}$. This is the non-slip velocity boundary condition where the fluid velocity at a boundary node is equal to the velocity of the boundary node itself.

STEP 2. Solve the pressure Poisson equation.

For interior nodes: solve Eq. (4c).

For solid boundary nodes: $\mathbf{n} \cdot \nabla p^{n+1} = \mathbf{n} \cdot (-\mathbf{a}_b + \frac{1}{Re} \nabla^2 \mathbf{u}^{n+1})$, where \mathbf{a}_b is the boundary node acceleration.

For far field boundary nodes: $\mathbf{n} \cdot \nabla p^{n+1} = 0$, which is again a Neumann-type boundary condition.

STEP 3. Compute the final velocity field \mathbf{u}^{n+1} .

For all computational nodes: solve Eq. (4b).

Above three steps are repeated until the solution converges. Any term containing the del operator (∇) or Laplace operator (∇^2) requires spatial discretization, the scheme depends on the type of computation nodes. Standard procedures can be followed for the 7-point central difference scheme, SVD-GFD scheme will be elaborated in next section. STEP 1 and STEP 3 are straightforward, where the calculation can be done separately on each node. STEP 2 is more complicated because pressure needs to be updated simultaneously to the next time step for all nodes. A large linear system in the form of $\mathbf{Ax} = \mathbf{b}$ is derived. \mathbf{A} is a sparse matrix containing the coefficients of the spatial discretization. \mathbf{x} is the pressure vector, the dimension of which is the total number of computation nodes. \mathbf{b} stores the results of $\frac{2}{\Delta t} \nabla \cdot \mathbf{u}^*$ and the boundary condition in STEP 2. In present study, the BICGStab method is adopted to solve the large linear system.

2.3 SVD-GFD Scheme

The generalized finite difference (GFD) method is based on the Taylor series expansion. Taylor series can represent an arbitrary function $f(\mathbf{x})$ as an infinite sum of terms calculated by the function's derivatives at a single point. Equation (5) shows the Taylor series expansion of $f(\mathbf{x})$ at point $\mathbf{x} = \mathbf{x}_0$ up to the n^{th} order. The value of the function at $\mathbf{x} = \mathbf{x}_0 + \Delta \mathbf{x}$ can be approximated by the Taylor series at $\mathbf{x} = \mathbf{x}_0$. In solving the NS equations, the function $f(\mathbf{x})$ can be the pressure field $p(\mathbf{x})$ or any velocity component $u(\mathbf{x}), v(\mathbf{x}), w(\mathbf{x})$.

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \sum_{1 \leq i_1+i_2+i_3 \leq n-1} \frac{\Delta x^{i_1} \Delta y^{i_2} \Delta z^{i_3}}{i_1! i_2! i_3!} \left[\frac{\partial^{i_1+i_2+i_3}}{\partial x^{i_1} \partial y^{i_2} \partial z^{i_3}} f \right]_{\mathbf{x}_0} + O(|\Delta \mathbf{x}|^n) \quad (5)$$

Second order spatial accuracy can be maintained if the Taylor series is truncated after the 4th order derivatives ($n = 4$). In this way, the first 19 derivatives are retained. If the values of function $f(\mathbf{x})$ are known on 19 surrounding nodes of \mathbf{x}_0 , we can obtain 19 equations about the derivatives $\frac{\partial^{i_1+i_2+i_3}}{\partial x^{i_1} \partial y^{i_2} \partial z^{i_3}} f$ at \mathbf{x}_0 , which forms a closed linear system for solving the derivatives. However, practice shows that the 19×19 matrix tends to be ill-conditioned, which is mainly caused by the poor spatial arrangement of the surrounding

nodes. A systematic nodal selection scheme is applied to find out the most suitable supporting nodes to form the equation system. Details of the scheme can be found in the work by Ang [17], Zhang [18] and Yao [19], which will not be elaborated here. The nodal selection scheme helps to improve the quality of the matrix but cannot eliminate the ill-conditioned problem. An over-determined algebraic system is used to solve this problem, which is formed by including more supporting nodes. Hence the final linear system is shown as follows, where $N > 19$ is the number of surrounding nodes.

$$\mathbf{S}_{N \times 19} \partial \mathbf{f}_{19 \times 1} = \Delta \mathbf{f}_{N \times 1} \quad (6)$$

$$\Delta \mathbf{f}_{N \times 1} = [f_1 - f_0 \quad f_2 - f_0 \quad \dots \quad f_N - f_0]^T$$

$$\mathbf{S}_{N \times 19} = \begin{bmatrix} \Delta x_1 & \Delta y_1 & \Delta z_1 & 0.5 \Delta x_1^2 & 0.5 \Delta y_1^2 & 0.5 \Delta z_1^2 & \dots & \Delta x_1 \Delta y_1 \Delta z_1 \\ \Delta x_2 & \Delta y_2 & \Delta z_2 & 0.5 \Delta x_2^2 & 0.5 \Delta y_2^2 & 0.5 \Delta z_2^2 & & \Delta x_2 \Delta y_2 \Delta z_2 \\ \vdots & & & & & & \ddots & \vdots \\ \Delta x_N & \Delta y_N & \Delta z_N & 0.5 \Delta x_N^2 & 0.5 \Delta y_N^2 & 0.5 \Delta z_N^2 & \dots & \Delta x_N \Delta y_N \Delta z_N \end{bmatrix}$$

$$\partial \mathbf{f}_{19 \times 1} = \left[\frac{\partial f(\mathbf{x}_0)}{\partial x} \quad \frac{\partial f(\mathbf{x}_0)}{\partial y} \quad \frac{\partial f(\mathbf{x}_0)}{\partial z} \quad \frac{\partial^2 f(\mathbf{x}_0)}{\partial x^2} \quad \frac{\partial^2 f(\mathbf{x}_0)}{\partial y^2} \quad \frac{\partial^2 f(\mathbf{x}_0)}{\partial z^2} \quad \dots \quad \frac{\partial^3 f(\mathbf{x}_0)}{\partial x \partial y \partial z} \right]^T$$

Pseudoinverse matrix of $\mathbf{S}_{N \times 19}$ needs to be computed to solve for $\partial \mathbf{f}_{19 \times 1}$, which is accomplished by the approach of single value decomposition (SVD).

2.4 Computation Acceleration

Above shows a brief introduction for the computational framework for current simulation, more details can be referred to in [16]. So far, we can see the numerical simulation is very computationally intensive due to the large mesh scale and time-dependent iterative FSI solver and thus parallelization is essential in the computation. Two parallelization techniques are used in current simulation, i.e., open multi-processing (OpenMP) and graphics processing units (GPU) acceleration. OpenMP is a shared-memory parallel architecture which provides great flexibility to achieve parallelization, where minor change is required on the code based on serial computation. General-purpose computing on graphics processing units (GPGPU) is an emerging heterogeneous computing technique that performs massive parallelization on graphics processing unit (GPU). This technology is designed to achieve high float point operation rates and is suitable for acceleration of numerically intensive CFD computations. Nowadays, a general-purpose GPU is designed with thousands of processing units to achieve high float-point operation rates, the structure of which is very suitable for current CFD simulation on a large grid system. The calculation of the projection method (BICGStab algorithm) with GPU acceleration is much faster than that with CPU parallelization, and the GPU acceleration becomes more advantageous as the grid amount increases [20]. The main drawback of the GPU acceleration is programming complexity, especially when extensive calculation is required on a single mesh grid, such as the singular value decomposition (SVD) procedure in current study. SVD calculation of the CFD solver was thus performed on CPUs with OpenMP parallel computation in present simulations. Hence, we combine the above two parallelization techniques to utilize their respective advantages.

3 Results and Discussion

The forward flight can be divided into two main consecutive stages, body pitching down and continuous acceleration. The first stage can be accomplished within few wing cycles and does not affect the flight stability much, while we have to stabilize the pitching angle in the second stage. Hence, the first step to fly forward is to pitch down so the aerodynamic force in the horizontal direction is increased and body drag is reduced.

The pitching down process is governed by two factors, aerodynamic moment and wing inertia-induced moment. The normal wing motion is restricted within the stroke plane for simple and stable flight control. While in the nature, insects apply the elevation angle very often, which results in a wide variety of wing kinematics. The oval-shape wing motion is adopted here for rapid pitch control, which is demonstrated in Fig. 2. Such wing motion is governed by the wing elevation function shown by Eq. (1b). The wing sweeps lower than the wing root during the downstroke and higher than the wing root during the upstroke, so the wing tip trajectory is an oval shape. Function of this type of elevation motion is shown by Eq. (7), where θ_a is prescribed as 10° during the pitching down process.

$$\theta = \theta_a \sin(2\pi t) \quad (7)$$

Both aerodynamic drag force and wing inertia force are in the opposite direction of wing motion. Wing drag/inertia force in downstroke has shorter moment arm than that in upstroke, which contributes a net pitching down aerodynamic/inertia-induced moment in a wingbeat. The magnitude of both moments increases with larger elevation amplitude. The aerodynamic moment superposes with the wing inertia-induced moment, and together they make the insect pitch down rapidly.

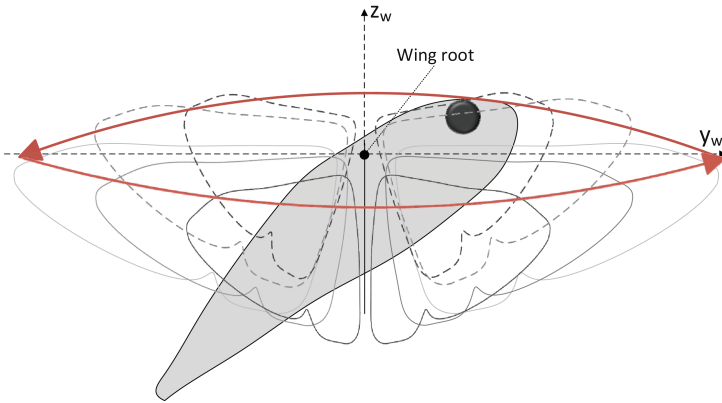


Fig. 2. Oval-shape wing motion

Since wing inertia-induced force is the internal force between the wings and body, the body pitch up (down) caused by wing mass does not impose pitching velocity or acceleration, governed by the law of conservation of angular momentum. However, that

is unavoidable in pitch motion due to the aerodynamic moment. Hence, orientation adjustment by wing inertia is theoretically more straightforward and easier to control.

After the fast pitching down stage, we need to maintain the designated pitching angle to ensure the insect fly forward stably. When the insect starts to gain speed, the pitching up moment induced by the flow field increases for any wing plane angle values, which can be seen from the prescribed motion study shown in Fig. 3. In that case, we maintain the oval wing motion to eliminate the induced moment. This was also discussed by Yao and Yeo [11].

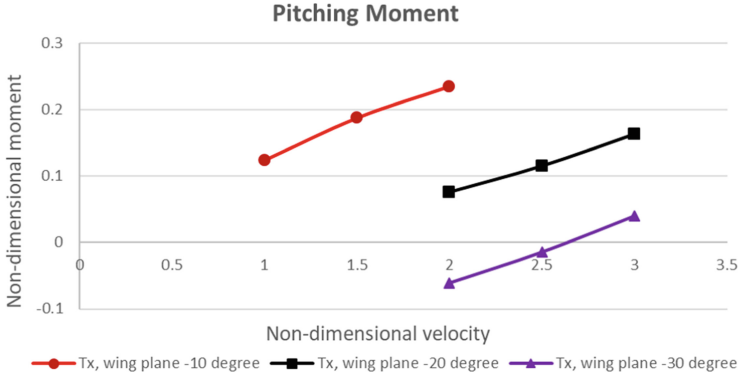


Fig. 3. Pitching moment variation with forward speed at different wing plane angles

As indicated by Fig. 3, the induced pitching up moment increases almost linearly with the speed at different wing plane angles. Therefore, we can simply relate the wing elevation amplitude and forward speed linearly as in Eq. (8).

$$\theta_a = k^\theta v \tag{8}$$

A constant linear coefficient may not be able to sustain a stable flight even though the pitching moment generated by oval wing motion may change linearly with elevation amplitude. The reason is because we cannot maintain the pitching angle at the designated value throughout the flight, and the actual induced moment may not follow exactly the trend shown in Fig. 3. Hence, we need modification to this linear relationship. Alternating linear coefficient can be used to counter the uncertain induced moment. If the insect tends to pitch up due to positive moment, we apply a larger coefficient to achieve a stronger pitching down effect. Similarly, we apply a smaller coefficient if it tends to pitch down. The alternating coefficient has advantage over a constant coefficient that it can stop the excessive moment from building up with a stronger recovery in time.

As analyzed previously, wing mass plays a significant role in pitch control during forward flight. To verify the analysis and find out the effect of wing mass, we simulated the forward flight in two situations, i.e., with or without wing mass. Since the effect of wing mass is mainly on the longitudinal motion, simulations reported in this paper do not take into consideration of the lateral dynamics to avoid the influence on each other. Figure 4 shows the comparison of forward velocity between flight with actual wing

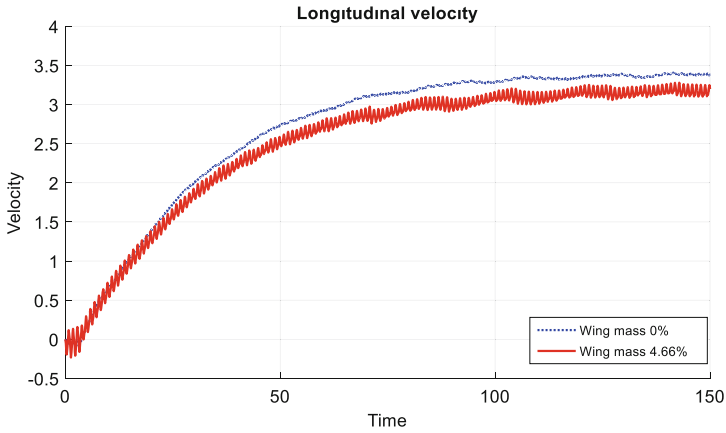


Fig. 4. Comparison of speed for forward acceleration with/without wing mass

mass ratio and that without wing mass, where the wingbeat frequency is fixed at 75 Hz. Clearly, we can find that the case without wing mass shows much smaller cyclic velocity oscillation. Same conclusion can be drawn for the pitching motion, which is presented

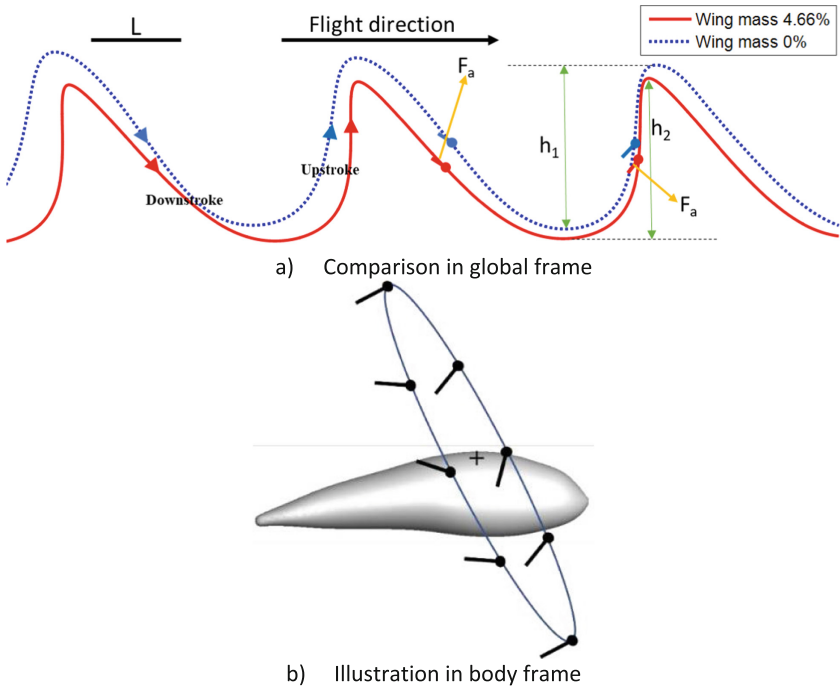


Fig. 5. Wing tip trajectories for forward acceleration with/without wing mass. Figures are based on actual flight data. Wing root is marked as (+) in figure. Wing chords show the approximate wing orientation with dotted leading edge. Unit wing length is represented by L.

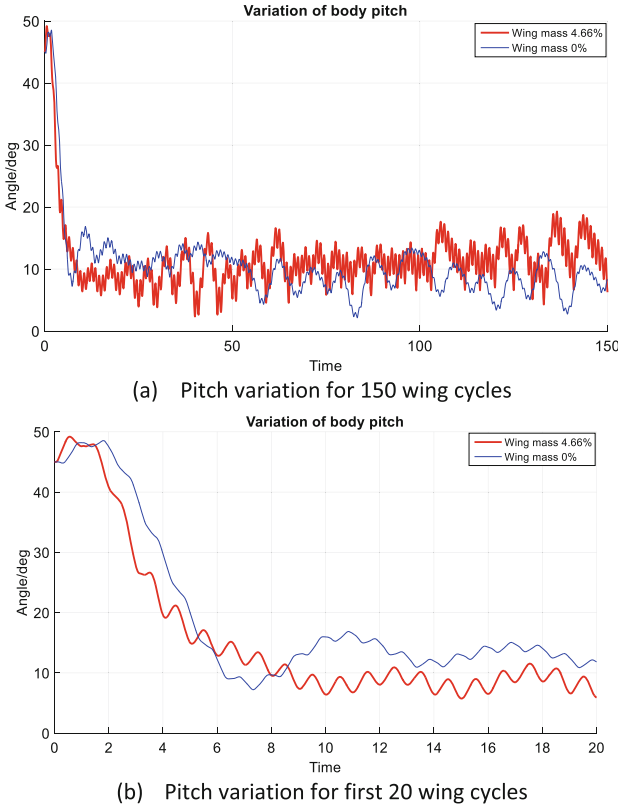


Fig. 6. Comparison of pitch for forward acceleration with/without wing mass

in Fig. 6. Hence, greater body oscillation is observed due to the wing inertia force. The flight with zero wing mass ratio accelerates faster and reaches a slightly higher speed in the end. The reason for that is because smaller body oscillation allows longer wing sweeping distance, which generates more aerodynamic forces. This can be verified by the wing tip trajectory at maximum speeds for both cases (see Fig. 5). The vertical wing sweeping distance is less dependent on the forward speed compared to the longitudinal sweeping distance, and we find that the positive longitudinal force is mainly generated in the upstroke, during which the wings move almost vertically viewed from global frame. Figure 5a shows that the flight without wing mass has a slightly longer range of wing tip trajectory in the vertical direction ($h_1 > h_2$) due to smaller body oscillation. Hence, more longitudinal thrust is generated for the case without wing mass, which contributes to faster acceleration and higher final speed.

Pitch control is more rapid and responsive for the flight with wing mass, as can be observed from Fig. 6a. The period of medium-term pitching oscillation for the case with wing mass is shorter than that without. This is because the wing inertia-induced moment gives rise to additional body rotation, which was explained previously. Another major difference is in the pitching down stage at the beginning of acceleration (first 3 wing

cycles). The flyer pitches down faster in the case with wing mass because this process is accomplished by both the aerodynamic moment and the wing inertia-induced moment, as shown in Fig. 6b. In addition, the residual pitching fluctuation after that is smaller since the internal wing inertia force does not change the pitching velocity. However, for the case without wing mass, the whole pitching down motion is contributed by aerodynamic moment alone, which can also change the pitching velocity.

4 Conclusion

The present paper investigated the effect of wing mass and wing elevation motion during insect free forward flight via numerical simulation. The numerical model integrated a Navier-Stokes flow solver with the Newtonian free-body dynamics of the model insect. Parallel computation is essential for current study, since the free flight simulation is computationally intensive due to the large mesh scale and the iterative solution for the FSI problem. Two parallelization techniques are used in current simulation, i.e., open multi-processing (OpenMP) and graphics processing units (GPU) acceleration. Two techniques are applied at the suitable computational algorithms respectively to take advantages of each. It is found that Oval-shaped wing elevating motion can help to generate large pitching down moment, both in terms of the aerodynamic moment and the wing inertia-induced moment. With such wing elevation motion, the flyer can quickly adjust its orientation for forward acceleration from the normal hovering status. On the other hand, wing mass tends to magnify such effect while prohibits the growth of pitching down velocity, which is favourable for the overall flight performance. Larger body oscillation is observed during forward flight for the case with actual wing mass compared to the case without wing mass. In addition, the forward flight with zero wing mass ratio accelerates faster and reaches a slightly higher speed in the end compared to the case with insect actual wing mass. The present study provides detailed information and access to the coupled dynamics of fluid and flyer in free flight condition. The present free flight model offers a prospective approach that could complement existing experiments with live insect subjects in a wider study of ‘real-time’ dynamics of insect flight and manoeuvres.

References

1. Osborne, M.: Aerodynamics of flapping flight with application to insects. *J. Exp. Biol.* **28**(2), 221–245 (1951)
2. Weis-Fogh, T., Jensen, M.: Biology and physics of locust flight. I. basic principles in insect flight. a critical review. *Phil. Trans. R. Soc. Lond. B* **239**(667), 415–458 (1956)
3. Weis-Fogh, T.: Energetics of hovering flight in hummingbirds and in drosophila. *J. Exp. Biol.* **56**(1), 79–104 (1972)
4. Weis-Fogh, T.: Quick estimates of flight fitness in hovering animals, including novel mechanisms for lift production. *J. Exp. Biol.* **59**(1), 169–230 (1973)
5. Ellington, C.: The aerodynamics of hovering insect flight. I. The quasi-steady analysis. *Philos. Trans. R. Soc. London. B Biol. Sci.* **305**(1122), 1–15 (1984)
6. Willmott, A.P., Ellington, C.P.: The mechanics of flight in the hawkmoth *manduca sexta*. I. Kinematics of hovering and forward flight. *J. Exp. Biol.* **200**(21), 2705–2722 (1997)

7. Willmott, A.P., Ellington, C.P.: The mechanics of flight in the hawkmoth *Manduca sexta*. II. Aerodynamic consequences of kinematic and morphological variation. *J. Exp. Biol.* **200**(21), 2723–2745 (1997)
8. Dickinson, M.H., Lehmann, F.-O., Sane, S.P.: Wing rotation and the aerodynamic basis of insect flight. *Science* **284**(5422), 1954–1960 (1999)
9. Fry, S.N., Sayaman, R., Dickinson, M.H.: The aerodynamics of free-flight maneuvers in *Drosophila*. *Science* **300**(5618), 495–498 (2003)
10. Wu, D., Yeo, K.S., Lim, T.T.: A numerical study on the free hovering flight of a model insect at low Reynolds number. *Comput. Fluids* **103**, 234–261 (2014)
11. Yao, Y., Yeo, K.S.: Longitudinal free flight of a model insect flyer at low Reynolds number. *Comput. Fluids* **162**, 72–90 (2018)
12. Yao, Y., Yeo, K.S.: *Manoeuvring Flight of a Model Insect—Saccadic Yaw and Sideslip*. *Computers & Fluids*, 2019
13. Yao, J., Yeo, K.S.: Free hovering of hummingbird hawkmoth and effects of wing mass and wing elevation. *Comput. Fluids* **186**, 99–127 (2019)
14. Yao, J., Yeo, K.: Forward flight and sideslip manoeuvre of a model hawkmoth. *J. Fluid Mech.* **896**, A22 (2020)
15. Wu, G., Zeng, L.: Measuring the kinematics of a free-flying hawk-moth (*Macroglossum stellatarum*) by a comb-fringe projection method. *Acta. Mech. Sin.* **26**(1), 67–71 (2010)
16. Yao, J., *Computational Aerodynamics of Hawkmoth Free Flight*. Thesis (2018)
17. Ang, S., et al.: A singular-value decomposition (SVD)-based generalized finite difference (GFD) method for close-interaction moving boundary flow problems. *Int. J. Numer. Meth. Eng.* **76**(12), 1892–1929 (2008)
18. Zhang, L.: *Unsteady aerodynamics of flapping wings*. Thesis (2013)
19. Yao, Y.: *Computational aerodynamics of flapping wing flight*. Thesis (2016)
20. Yao, Y., Yeo, K.-S.: An Application of GPU Acceleration in CFD simulation for insect flight. *Supercomput. Front. Innov.* **4**(2), 13–26 (2017)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Exploring the Dynamics of Quantum Information in Many-Body Localised Systems with High Performance Computing

Shao-Hen Chiew^{1,2(✉)}, Leong-Chuan Kwek^{2,3,4}, and Chee-Kong Lee⁵

¹ Department of Physics, Faculty of Science, National University of Singapore, Blk S12 Level 2, Science Drive 3, Singapore 117551, Singapore

shaohenc@gmail.com

² Centre for Quantum Technologies, National University of Singapore, Singapore 117543, Singapore

³ MajuLab, CNRS-UNS-NUS-NTU International Joint Research Unit, Singapore UMI 3654, Singapore, Singapore

⁴ National Institute of Education, Nanyang Technological University, Singapore 637616, Singapore

⁵ Tencent America, Palo Alto, CA 94306, USA

Abstract. Conventional many-body quantum systems thermalize under their own dynamics, losing information about their initial configurations to the environment. However, it is known that a strong disorder results in many-body localization (MBL). A closed quantum systems with MBL retains local information even in the presence of interactions. Here, we numerically study the propagation and scrambling of quantum information of a closed system in the MBL phase from an information theoretic perspective. By simulating the dynamics and equilibration of the temporal mutual information for long times, we see that it can distinguish between MBL and ergodic phases.

Keywords: Quantum dynamics · Quantum information · Disordered systems

1 Introduction

The fate of a generic many-body quantum system can be described by quantum statistical mechanics at equilibrium, where it is expected that it eventually thermalizes through the process of thermal equilibration regardless of the system's initial state. Recently, however, it has become clear that there exists exceptional disordered quantum system that can avoid this fate through localization [1–3]. This phenomenon, termed Many-Body Localisation (MBL), leads to a plethora of interesting features that cannot be described by quantum statistical mechanics, including the preservation of information, the slow spreading of entanglement, the emergence of integrability, and so forth [4].

© The Author(s) 2022

D. K. Panda and M. Sullivan (Eds.): SCFA 2022, LNCS 13214, pp. 43–58, 2022.

https://doi.org/10.1007/978-3-031-10419-0_4

Concomitant to the development of a new understanding of the MBL phase, there has also been significant progress in the experimental simulation of quantum many-body systems [5–10]. With a greater degree of tunability, control, manipulation and isolation from the environment, such systems provide a perfect avenue towards a greater understanding of strongly-correlated many-body quantum systems. Examples of such experimental setups are ultracold atoms in optical lattices, trapped ions, and nuclear and electron spins associated with impurity atoms in diamond nitrogen-vacancy centers [4]. These systems have also been studied through the lens of quantum information science using concepts and tools such as quantum entanglement, quantum coherence and the quantum Fisher information. For example, entanglement growth can be detected through suitable witnesses or with the quantum Fisher information [11]. The latter provides a lower bound on the entanglement in the system with just a measurement of two-body correlators, which can be efficiently accessed with site-resolved imaging [6]. In some cases, for instance in ion traps, partial or full quantum state tomography can be performed [12].

Systems in the MBL phase differs significantly from systems in the ergodic phase in many aspects. From an experimental perspective, the equilibration of physical observables to non-thermal values at long times [13, 14] is interesting as the expectation values of physical observables such as local magnetisation can be experimentally probed [5–7]. However, local observables can only reveal part of the complete picture: to fully investigate MBL, it is fruitful to resort to more abstract quantities that capture finer details from the information-theoretic properties of the MBL phase. This is precisely the approach that we will take in this paper.

In particular, we focus on understanding the localization and the propagation of information as the MBL systems evolve. We begin with a brief overview of relevant concepts and methods in the next section. We then proceed to discuss the main problem in detail, and present our numerical results on the temporal mutual information at length in Sect. 3.

2 Background and Numerical Setup for Dynamics

From an information-theoretic perspective, a hallmark of the MBL phase is the logarithmic spreading of the entanglement entropy. The *entanglement entropy* between two subsystems of a quantum state ρ , or alternatively the Von Neumann entropy of the reduced density matrix for either subsystem, is defined by:

$$S_{ent}(\rho_A) = -\text{Tr}(\rho_A \log \rho_A), \quad (1)$$

where $\rho_A = \text{Tr}_B(\rho_{AB})$, with A being a subsystem of the total system AB . It measures the extent to which the subsystems A and B are entangled with one another. Starting with an initial non-thermal product state, one can show [15] that the entanglement entropy of a MBL system grows logarithmically, i.e.:

$$S_{ent}(\rho_{MBL}) \propto \log(t), \quad (2)$$

in contrast to the ballistic spread of entropy in the ergodic case:

$$S_{ent}(\rho_{erg}) \propto t. \quad (3)$$

This indicates much slower spreading of entanglement in the MBL phase. Indeed, this slow spreading of correlations is often regarded as one of the distinguishing features of MBL from ergodic systems.

Another relevant quantity is the quantum mutual information (QMI). For a system partitioned into two subsystems A and B , the QMI between A and B is defined as:

$$I(A : B) = S(\rho_A) + S(\rho_B) - S(\rho_{AB}), \quad (4)$$

where $S(\rho)$ is the entanglement entropy of the state ρ as defined in the previous paragraph. It measures the total correlations shared between the two subsystems, or equivalently how much information is gained about one subsystem by measuring the state of the other. The separability of two subsystems, i.e. $\rho_{AB} = \rho_A \otimes \rho_B$ implies zero QMI, while non-zero QMI implies non-zero correlation or entanglement. The growth and equilibration of the QMI can be used as a diagnostic tool for the MBL phase [16, 17]. Numerical results indicate that the QMI in an MBL phase is exponentially localized in space, which is consistent with our intuition of a localized phase.

2.1 Physical Model

An important model that exhibits MBL is the 1D isotropic Heisenberg spin-1/2 chain, subject to random transverse magnetic fields [16, 17]. For an array of L spins obeying open boundary conditions, the Hamiltonian for this system is given by:

$$H = J \sum_{i=1}^{L-1} \mathbf{S}_i \cdot \mathbf{S}_{i+1} + \sum_{i=1}^L h_i S_i^z. \quad (5)$$

Here, $\mathbf{S}_i = (S_i^x, S_i^y, S_i^z)$ is the vector of local spin operators at site i , with $i \in [1, L]$, J the interaction strength, and h_i the strength of the disordered magnetic field at site i , which is a random real number uniformly distributed in the interval $[-W, W]$. This well-studied system is known to exhibit MBL, with an ergodic-MBL transition occurring at $W \approx 3.5J$ [2, 18, 19]. In the following sections, we will focus exclusively on this model, with the disorder parameter W controlling the system's localization. Throughout the article and in numerical simulations, we will also set $J = 1$ consistently.

2.2 Simulation of Unitary Dynamics

To study the steady-state properties of a closed system, one can perform a *quantum quench*, where an initial nonequilibrium state $|\psi(0)\rangle$ (which is the ground state of a Hamiltonian H_0) is first prepared, and evolved under the unitary dynamics of another Hamiltonian H according to Schrödinger's equation:

$$i\hbar \frac{\partial}{\partial t} |\psi(t)\rangle = H |\psi(t)\rangle. \quad (6)$$

$|\psi(t)\rangle$ can then be studied either experimentally by measuring the values of physical observables in a physical setup, or numerically by simulating less experimentally accessible quantities such as the entanglement entropy and the QMI. Of particular interest to us are signatures that manifest when MBL systems evolve in time.

We simulate the unitary dynamics of a closed quantum system by directly integrating Schrödinger's equation for a time independent Hamiltonian H , Eq. (6). In a particular basis $|\phi_k\rangle$, we have:

$$|\psi(t)\rangle = \sum_k c_k(t) |\phi_k\rangle. \quad (7)$$

The set of coupled first-order differential equations then takes the form:

$$i\hbar \frac{\partial c_k(t)}{\partial t} = \sum_i [H]_{ki} c_i(t), \quad (8)$$

which can be readily integrated by an ODE solver to yield $|\psi(t)\rangle$.

Importantly, to study disordered systems such as Eq. (5), we study the averages of quantities over multiple *disorder realizations*, with each realisation corresponding to a randomly sampled set of transverse magnetic fields h_i . The number of realizations range from 100–1000 in this project.

The numerical simulation of quantum dynamics is implemented in Python with QuTiP [20], an open-source software for simulating the dynamics of closed and open quantum systems. In particular, the Complex-valued Variable-coefficient Ordinary Differential Equation (zvode) solver [21] is used to integrate Eq. (6). Time intensive simulations are also performed with the High Performance Computing (HPC) clusters from NUS and NSCC.

3 Information Scrambling and Delocalization in MBL Systems

An important characteristics of MBL is the slow propagation of quantum information. Along these lines, we wish to understand how an initially localized information is spatially spread across a many-body quantum system under time evolution, and how the phenomenon of many-body localization changes the answer to this question.

These considerations lead us to the notion of *information scrambling*, which is the spreading of local information across many-body quantum systems, such that they can only be recovered by non-local measurements. Related to thermalization and chaos, the notion of scrambling has been used recently to study the quantum information of black holes [22, 23], and can be experimentally probed [24]. Naturally, it is interesting to relate this notion to the MBL phase, given its information-localizing nature.

In this section, we relate a proposed measure of scrambling, the *temporal mutual information* (following [25]), with the MBL phase, and investigate its

qualitative differences with the ergodic phase with numerical and some analytical arguments. We begin with a brief review on the channel-state duality and use it to define the temporal mutual information.

3.1 Channel-State Duality

Consider the action of a unitary operator¹ $U(t)$ that acts on vectors in \mathcal{H} , written in a basis as:

$$U(t) = \sum_{i,j} u_{ij} |i\rangle \langle j|. \quad (9)$$

This operator is then isomorphic to a state in $\mathcal{H} \otimes \mathcal{H}$:

$$|U(t)\rangle = \sum_{i,j} u_{ij} |i\rangle |j\rangle, \quad (10)$$

an isomorphism known as the *channel-state duality* (or the Choi-Jamiołkowski isomorphism). More generally, consider an arbitrary input ensemble $\rho_{in} = \sum_i p_i |\psi_i\rangle \langle \psi_i|$. Each state $|\psi_i\rangle$ in this statistical ensemble evolves into $|\phi_i\rangle = U(t)|\psi_i\rangle$, so that the entire ensemble becomes $\rho_{out} = \sum_i p_i |\phi_i\rangle \langle \phi_i|$. The action of $U(t)$ on the input state can then be summarized by the pure state:

$$|\Psi\rangle = \sum_j \sqrt{p_j} |\psi_j\rangle_{in} \otimes |\phi_j\rangle_{out} = \mathbb{1} \otimes U(t) \sum_j \sqrt{p_j} |\psi_j\rangle_{in} \otimes |\psi_j\rangle_{out}. \quad (11)$$

$|\Psi\rangle$ contains all information about the action of $U(t)$ on ρ_{in} . In particular, we have:

$$\rho_{in} = \text{Tr}_{out}(|\Psi\rangle \langle \Psi|), \quad (12)$$

$$\rho_{out} = \text{Tr}_{in}(|\Psi\rangle \langle \Psi|). \quad (13)$$

Importantly, the state in the form of Eq. (11) treats the input and output states on equal footing. If we consider the unitary operator to be the propagator $U(t) = e^{-iHt}$ of the Hamiltonian H , Eq. (11) then contains information about the the state at different times, before and after the evolution due to $U(t)$.

3.2 Temporal Mutual Information

For concreteness, consider a 1D lattice of spins in $(\mathcal{H})^{\otimes N}$ that is evolving under a Hamiltonian H . The state $|\Psi\rangle$ dual to the channel $U(t)$ then lives in $\mathcal{H}_{in} \otimes \mathcal{H}_{out}$, where $\mathcal{H}_{in} = \mathcal{H}_{out} = (\mathcal{H})^{\otimes N}$. We partition \mathcal{H}_{in} arbitrarily into subsystems A and B , and \mathcal{H}_{out} into C and D (Fig. 1 represents the situation schematically).

¹ In general, the evolution need not be unitary, but can be a quantum channel (i.e. a trace-preserving completely positive map) in the case where the state is an open subsystem of a larger, closed system. We will only consider unitary channels in the following analyses.

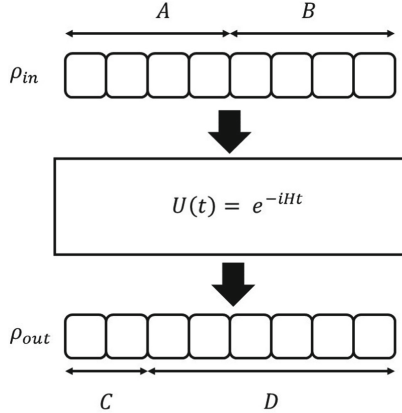


Fig. 1. Schematic representation of the spatial partition of input 1D lattice into A and B , and the output state into C and D after a unitary evolution generated by H . In general, there need not be an equal number of partitions of the input and output states, and A and C (B and D) need not correspond to the same spatial partitions. If they do, we denote them $A = A(0)$ and $C = A(t)$ ($B = B(0)$ and $D = B(t)$).

Following [24], one can then define the entanglement entropy between subsystems at different times and different spatial sites by tracing out appropriate subsystems from the dual state $\rho = |\Psi\rangle\langle\Psi|$. For example,

$$S(\rho_{AC}) \equiv -\text{Tr}(\rho_{AC} \log \rho_{AC}), \quad (14)$$

where $\rho_{AC} = \text{Tr}_{BD}(\rho)$ is the reduced density matrix containing subsystem A before the unitary evolution and C after the evolution. We can further define a more useful quantity, which is the mutual information at different times and different spatial sites:

$$I(A : C) \equiv S(\rho_A) + S(\rho_C) - S(\rho_{AC}). \quad (15)$$

This quantity, which we refer to as the *temporal mutual information*, intuitively quantifies the amount of information that one can obtain about subsystem A by measuring subsystem C at a later time. Furthermore, if A and C correspond to the same sites spatially (at different times), which we denote as $A \equiv A(0)$ and $C \equiv A(t)$, $I(A(0) : A(t))$ quantifies information contained in a spatial region ($A(t)$) about the same region before time evolution ($A(0)$) - in other words how much information can still be extracted from a region of space about its past configuration.

3.3 Problem Statement

We can study the delocalization of information using a game played between Alice and Bob. Suppose Alice has a source of classical information $X = 0, \dots, N$

with probability distribution p_0, \dots, p_N , which she chooses to encode in a set of quantum states $\{|\psi_0\rangle, \dots, |\psi_N\rangle\}$. Alice prepares a state $|\psi_X\rangle$ from this set and sends it to Bob, so the state that Bob effectively studies is:

$$\rho = \sum_{i=0}^N p_i |\psi_i\rangle\langle\psi_i|. \quad (16)$$

Bob's task is then to determine the index X based on his knowledge about the state of some part of the system.

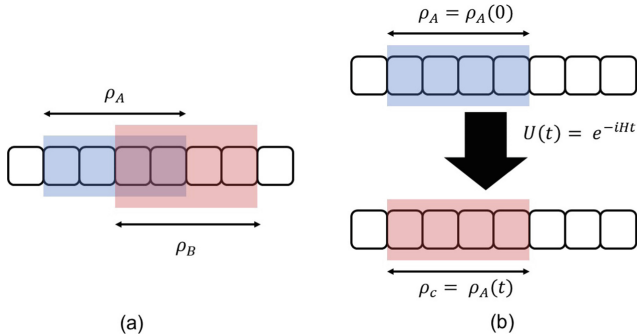


Fig. 2. Schematic of the game between Alice and Bob, realized on a 1D lattice of spins. ρ_A is the information-bearing state provided by Alice, and ρ_B is the state that Bob uses to infer ρ_A . In (a), Bob infers ρ_A using a state ρ_B at the same time, so the information that Bob can extract is $I(A : B)$. In (b), Bob infers $\rho_A = \rho_A(0)$ using a state at a later time, ρ_C , which happens to be the same spins after the evolution, i.e. $\rho_C = \rho_A(t)$. The information that Bob can extract is then the temporal mutual information $I(A(0) : A(t))$. In general, ρ_C need not be the same spins. Our analyses will focus solely on the temporal mutual information.

For concreteness, suppose further that these quantum states are realized on part of a 1D lattice of spins, i.e. $\rho_A \in \mathcal{H}^A = (\mathcal{H})^{\otimes K} \subset (\mathcal{H})^{\otimes L}$, with $K < L$, so that $(\mathcal{H})^{\otimes L}$ constitutes the system (See. Fig. 2a). Calling A the partition that contains the information-bearing state ρ_A , the information that Bob can extract on ρ_A if he has full knowledge of a state ρ_B at another partition B is then given by the quantum mutual information:

$$I(A : B) = S(\rho_A) + S(\rho_B) - S(\rho_{AB}), \quad (17)$$

as discussed in Sect. 1.

Here, ρ_A and ρ_B correspond to states at the same time - this is the approach taken by [16, 17] to study MBL - while Eq. (15) from the previous section provides an extension to states at different times. Our question can now be rephrased as the following:

How much information can Bob obtain about an initial state ρ_A if he has knowledge about a state ρ_C after the action of a unitary evolution $U(t)$?

Our following investigations will focus on determining the behaviour of $I(A(0) : C(t))$ for different partitions $C(t)$. When $C = A(t)$, $I(A(0) : A(t))$ measures the amount of information that remains in the original subsystem after time evolution. On the other hand, if $C(t)$ is spatially disjoint from $A(0)$, $I(A(0) : C(t))$ measures the amount of information that has “leaked out” to another region C outside of A (See. Fig. 2b). We expect the behaviour of $I(A(0) : C(t))$ to differ depending on whether the lattice is ergodic or localized, and in our following work we obtain numerical evidence that our intuition is indeed true.

3.4 Numerical Results

We consider the closed system as consisting of 6 spins, where the first two spins are the information bearing spins. This closed system evolves under the Heisenberg XXX model Hamiltonian Eq. (5) that has been discussed in the previous sections, with the disorder parameter W controlling the strength of localization.

We encode the classical information source $X = 0, 1, 2, 3$ in orthogonal states $\{|\uparrow\uparrow\rangle, |\uparrow\downarrow\rangle, |\downarrow\uparrow\rangle, |\downarrow\downarrow\rangle\}$ with equal probabilities $p_0 = p_1 = p_2 = p_3$, so that the information bearing state $\rho_A = \sum_i p_i |\psi_i\rangle\langle\psi_i|$ corresponds to the maximally mixed state $\rho_A = \frac{1}{4}\mathbb{1}$. It is embedded in an environment (the 4 remaining spins) which we take to be the Néel state $|\uparrow\downarrow\uparrow\downarrow\rangle$. The combined initial state is thus:

$$\rho = \frac{1}{4}\mathbb{1} \otimes |\uparrow\downarrow\uparrow\downarrow\rangle. \quad (18)$$

With $U(t) = e^{-iHt}$, we can then use the channel-state duality to construct the pure state $|\Psi\rangle$ from Eq. (11), and monitor the evolution of the temporal mutual information $I(A(0) : C(t))$ with different partitions $C(t)$ and disorder W .

Dynamics of Initially Localized Information. Starting from ρ , how does information that is initially localized in ρ_A leak out to its surroundings after time evolution due to $U(t)$? This can be monitored by following the evolution of $I(A(0), A(t))$ - from an initially maximal value, it is expected to decay as time progresses, indicating that information initially contained in $A(0)$ has decayed.

Figure 3a shows the evolution of $I(A(0), A(t))$ for different values of W . This decay can indeed be noticed, with the final steady-state value depending on the disorder strength. In addition, we can also choose $C(t)$ to be spatially disjoint with $A(t)$. $I(A(0), C(t))$ then monitors information initially contained in $A(0)$ that has leaked to another spatially disjoint region $C(t)$. This is shown in Fig. 3b. Both plots agree with our expectation that localization strength directly affects the amount of information about the initial state that has leaked out spatially to other regions.

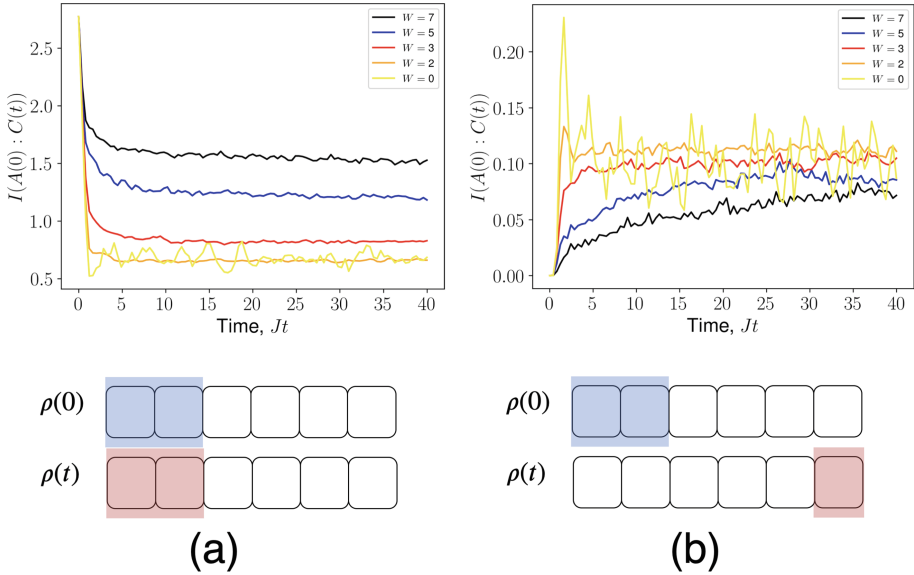


Fig. 3. Evolution of $I(A(0) : C(t))$ for different partitions $C(t)$ and disorder strength W (indicated by different colors), averaged over 50 disorder realizations. In (a), $C(t)$ is chosen to correspond to the same region A . $I(A(0) : C(t))$ is then observed to decay and equilibrate to a lower steady-state value that depends on the strength of localization, controlled by increasing W . In (b), $C(t)$ is disjoint from A , and chosen to be the furthest single site away from A . While initially containing no information about $A(0)$, information leaks out into this region as the system evolves and becomes entangled to the environment. We also graphically indicate the partitions below the plots, with blue marking partition A and red marking C of the temporal mutual information $I(A : C)$.

Having qualitatively shown the effects of localization strength on $I(A(0), C(t))$, we repeat this simulation more fully with all possible future partitions $C(t)$ for the ergodic case with $W = 0.1J$ and the MBL case with $W = 7J$, shown in Fig. 4. There are $2^n - 1 = 31$ number of ways to choose $C(t)$, corresponding to the 31 curves in Fig. 4. Several noteworthy observations can be extracted from Fig. 4:

- Similar to the previous plots, the steady-state values of $I(A(0) : C(t))$ depends on W . (This scaling is investigated more thoroughly in the next subsection)
- I increases as the size of the partition $C(t)$ is increased. This follows from the monotonicity of the quantum mutual information, i.e. $I(X : YZ) > I(X : Y)$.
- In the MBL phase, the steady-state values of $I(A(0) : C(t))$ decays with distance from the initial 2 spins. (For example, $I(A(0) : [5]) > I(A(0) : [6]) > \dots > I(A(0) : [9])$). The ergodic phase, on the other hand, does not exhibit this

spatial decay² ($I(A(0) : [6]) = \dots = I(A(0) : [9])$). This possibly demonstrates that in the ergodic phase, initially localized information is evenly distributed across the entire chain, while in the MBL phase the distribution of information depends on distance from the first 2 spins.

- At short times, the growth/decay of I occurs more rapidly in the ergodic phase, compared to the MBL phase. This can be attributed to the slow spreading of entanglement in the MBL phase, compared to the ballistic spread in the ergodic phase.
- $I(A(0) : [56789](t))$ is constant and equal to $2S(\rho_A(0)) = 2 \ln 4 \approx 2.77$. This is due to identity Eq. (20), representing the conservation of information, which we prove in the last subsection.
- Every curve (Except $I(A(0) : [56789](t))$) has a symmetric counterpart such that the sum between these two curves at any time is $2S(\rho_A(0))$. (For example, $I(A(0) : [5](t)) + I(A(0) : [6789](t)) = 2S(\rho_A(0))$) This can be explained with the identity Eq. (22) in the final subsection.

Steady-State Values of Temporal Mutual Information as a Function of Localization Strength. From the dynamics above, $I(A(0), C(t))$ reaches steady-state values after evolving for $t \approx 15J$. Choosing the final 20% of the evolution as the steady-state window t_{SS} , we define the steady-state TMI as:

$$\bar{I}(A(0), C) = \frac{1}{\Delta t_{SS}} \int_{t_{SS}} I(A(0), C(t)) dt. \quad (19)$$

In this section, we investigate these steady-state values as a function of disorder/localization strength W . The results are shown in Fig. 5. We note the following observations:

- Indeed, the value of $\bar{I}(A(0) : C)$ for any partition C that contains $A = [5]$ increases with W , signalling increasing localization of information in the initial spatial region. On the contrary, the value of $\bar{I}(A(0) : C)$ for any partition that does not contain $[5]$ decreases with W , as less information has leaked out to these partitions.
- As W is increased so that the system transitions into the MBL phase, the decay of $\bar{I}(A(0) : C)$ as C is chosen to be further away from the first 2 spins (also observed and discussed in the previous section) is again visible from the splitting of initially coinciding lines. Again, this indicates that ergodicity spreads information throughout the system evenly, while MBL tends to localize information with a strength that decays spatially.

² The large value of $I(A(0) : [5])$ compared to $I(A(0) : [6])$, ..., $I(A(0) : [9])$ is due in part to the index $[5]$ representing 2 spins.

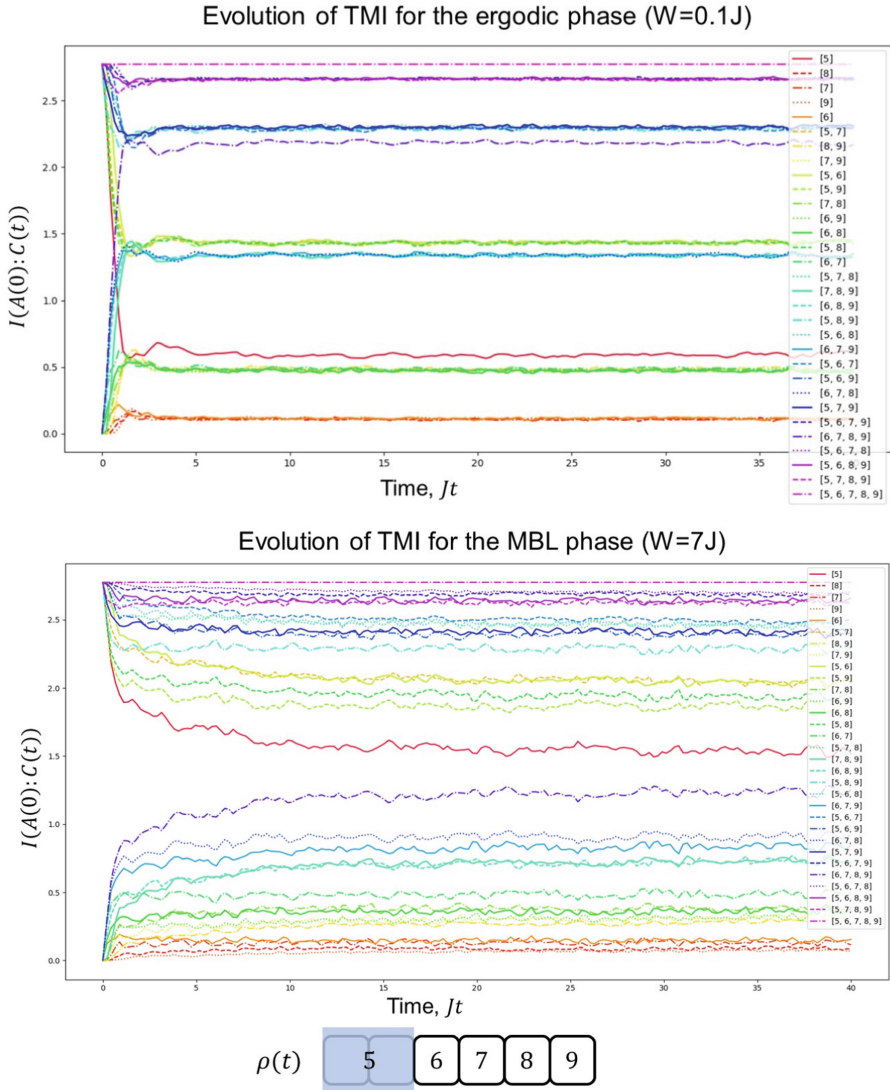


Fig. 4. Evolution of temporal mutual information for all possible partitions of $C(t)$, for the ergodic phase (Top plot) and the MBL phase (Bottom plot). Colors represent different choices of $C(t)$ and are labelled in the legend, and the indices corresponding to different sites are shown in the schematic below the plots (Blue highlighting indicates information-bearing spins). For example, $I(A(0), [5](t))$ is the temporal mutual information between $A(0)$ and $A(t)$, $I(A(0), [56](t))$ is the temporal mutual information between $A(0)$ and the first 3 spins at time t , and $I(A(0), [56789](t))$ is the temporal mutual information between $A(0)$ and the entire system plus environment at time t .

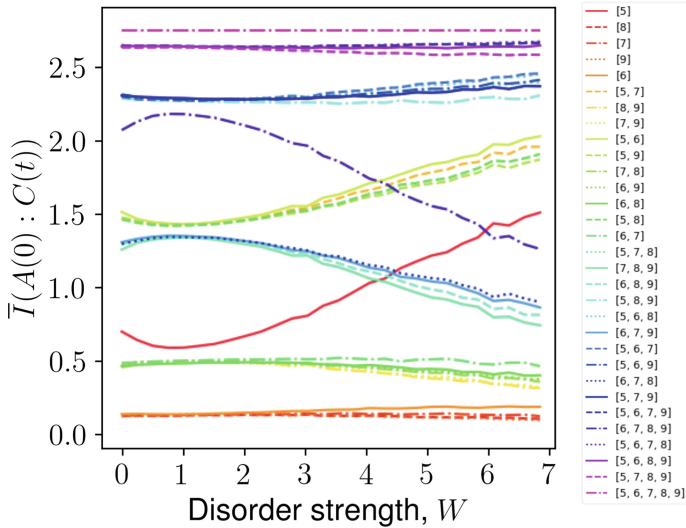


Fig. 5. $\bar{I}(A(0) : C)$ as a function of disorder strength W for all possible partitions C . Colors represent different choices of C and are labelled in the legend, and the indices corresponding to different sites can be found in the schematic below Fig. 4. Each value of \bar{I} is obtained by averaging over 500 disorder realizations, and the averaging window is chosen to be the final 20% of a total evolution time of $Jt = 40$.

Scaling with System Size. To study how the steady state temporal mutual information $\bar{I}(A(0) : C)$ scales with system size L , and whether it is useful for detecting the location of the critical disorder W_c at which the ergodic-MBL transition occurs, we perform additional simulations with different values of L .

Similar analyses [26, 27] that study the ergodic-MBL transition for finite system sizes provide evidence that the entanglement entropy and Holevo quantity can help locate the the ergodic-MBL transition that occurs at the thermodynamic limit, $L \rightarrow \infty$, where these quantities vary discontinuously across a critical disorder strength W_c . If the behaviour of $\bar{I}(A(0) : C)$ against W approaches a step function in a similar manner as the system size L tends to infinity, the location of the discontinuity then marks the location of the critical disorder W_c .

The results of some preliminary investigations for the same Heisenberg XXX spin chain³ are presented in Fig. 6 for $L = 4, 6, 8, 10, 12$. From the figure, while there are good indications that the curves are converging to a sigmoidal curve as L increases, suggesting a possible scaling law for $\bar{I}(A(0) : C)$, the limitations of our numerics prevent more concrete claims. More samples and larger system sizes will be needed to establish a scaling law.

³ The chosen configuration in this section is slightly different from the previous sections. Here, half of the system carries information, while in the previous sections only 2 out of 6 spins do.

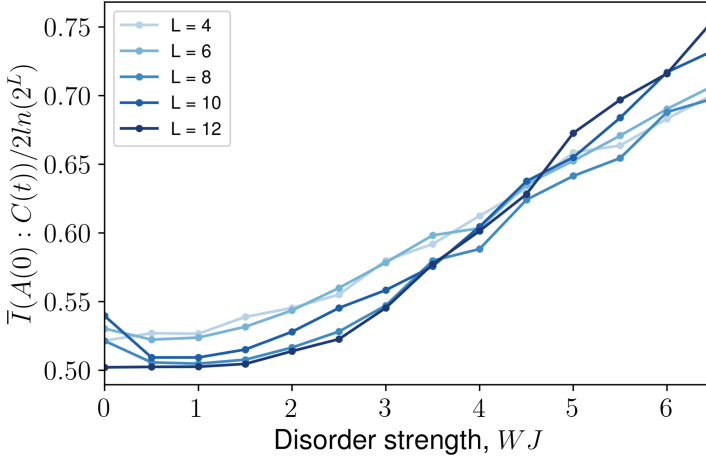


Fig. 6. Normalized steady-state values of $I(A(0) : C)$ against disorder, for system sizes $L = 6, 8, 10, 12$. The information-bearing state consists of the first $L/2$ spins, while the environment constitute the remaining $L/2$ spins. $\bar{I}(A(0) : C)$ is normalized with $2S(A(0)) = 2\ln(2^L)$ so that the same y-scale can be used to compare \bar{I} across different system sizes. Each point is produced by averaging over 100–200 disorder realizations, with an averaging window chosen to be the final 20% of a total evolution time of $Jt = 40$.

Mathematical Identities Finally, we state and prove a few identities on the temporal mutual information that explains some features of the above numerical results. Let L symbolically denote the system, and $\{A, B\}$ a partition of L . Suppose further that $\rho_A(0)$ is a mixed state and $\rho_B(0)$ is a pure state, so that $\rho(0) = \rho_A(0) \otimes \rho_B(0)$ (corresponding to our setup above). Recall that $\rho = |\Psi\rangle\langle\Psi|$ is the (pure) dual state obtained using the channel-state duality Eq. (11).

Lemma 1.

$$I(A(0) : L(t)) = 2S(\rho_A(0)). \quad (20)$$

Information initially contained in a subsystem A can always be fully re-extracted at a later time from the full system L .

Proof. From the definition of the temporal mutual information Eq. (15),

$$I(A(0) : L(t)) = S(\rho_{A(0)}) + S(\rho_{L(t)}) - S(\rho_{A(0)L(t)}). \quad (21)$$

We use the fact that the entropy of the bipartitions of a pure state are equal. Since the dual state ρ is pure, this implies that the second term is $S(\rho_{L(t)}) = S(\rho_{L(0)}) = S(\rho(0)) = S(\rho_A(0) \otimes \rho_B(0))$. The same fact yields $S(\rho_{A(0)L(t)}) = S(\rho_{B(0)})$ for the third term. Finally, $\rho_B(0)$ being pure implies that the third term vanishes, while the second term becomes $S(\rho_{L(t)}) = S(\rho_A(0))$, leading to the result.

Lemma 2. *If $\{C, D\}$ is an arbitrary partition of L , then:*

$$I(A(0) : L(t)) = I(A(0) : C(t)) + I(A(0) : D(t)). \quad (22)$$

Proof. Firstly, note that the subsystem $B(0)$ can always be traced out without changing the entropy. This can be seen by applying the triangle inequality and the subadditivity of the entropy on any subsystem $B(0)X$ containing $B(0)$:

$$|S(\rho_{B(0)}) - S(\rho_X)| \leq S(\rho_{B(0)X}) \leq S(\rho_X) + S(\rho_{B(0)}). \quad (23)$$

$\rho_{B(0)}$ being pure then implies that $S(\rho_{B(0)X}) = S(\rho_X)$.

The right hand expression is by definition:

$$\begin{aligned} I(A(0) : C(t)) + I(A(0) : D(t)) &= S(\rho_{A(0)}) + S(\rho_{C(t)}) \\ &\quad - S(\rho_{A(0)C(t)}) + S(\rho_{A(0)}) + S(\rho_{D(t)}) - S(\rho_{A(0)D(t)}). \end{aligned} \quad (24)$$

Some terms can be simplified:

$$S(\rho_{A(0)D(t)}) = S(\rho_{B(0)C(t)}) = S(\rho_{C(t)}) \quad (25)$$

$$S(\rho_{A(0)C(t)}) = S(\rho_{B(0)D(t)}) = S(\rho_{D(t)}), \quad (26)$$

where the first equality is because the entropies of the bipartitions of a pure state are equal, and the second equality results from our initial note. Finally, cancelling some terms yield:

$$I(A(0) : C(t)) + I(A(0) : D(t)) = 2S(\rho_{A(0)}) = I(A(0) : L(t)), \quad (27)$$

where the second equality is due to the previous identity Eq. (20).

Lemma 3. *We have:*

$$I(A(0) : A(t)) = S(\rho_{A(0)}) + S(\rho_{A(t)}) - S(\rho_{B(t)}). \quad (28)$$

Proof. Note that:

$$S(\rho_{A(0)A(t)}) = S(\rho_{B(0)B(t)}) = S(\rho_{B(t)}), \quad (29)$$

where the first equality is because the entropies of the bipartitions of a pure state are equal, and the second equality results from tracing out $B(0)$ not affecting the entropy. Applying the definition of $I(A(0) : A(t))$ then leads to the result.

4 Conclusion and Discussion

We have investigated information localization in MBL systems from an information-theoretic perspective by introducing the temporal mutual information. Unlike the quantum mutual information, which depends on states at the same time, the temporal mutual information allows us to monitor the spread of information as the system evolves. From our numerical simulations on its

dynamics, we observe that its evolution and steady-state behavior agree with our intuition that the MBL phase should localize and slow the spread of information. We also find some preliminary indications that it can be useful in identifying the ergodic-MBL transition, based on its scaling with system size.

Further work in this direction should constitute a better understanding of the temporal mutual information, and its relation with the entanglement entropy. In light of Eq. (28), there is a simple relation between I and S ; does the temporal mutual information then contain more information, or is the entanglement entropy sufficient in characterising the spread of information? Otherwise, the scaling behaviour of I could be investigated more thoroughly with larger system sizes. This would ideally involve better suited computational techniques such as the use of matrix product states and the time-evolving block decimation algorithm.

Acknowledgments. KLC is supported by the Ministry of Education and the National Research Foundation under the Center for Quantum Technologies (CQT).

References

1. Serbyn, M., Papić, Z., Abanin, D.A.: Local conservation laws and the structure of the many-body localized states. *Phys. Rev. Lett.* **111**(12), 127201 (2013)
2. Huse, D.A., Nandkishore, R., Oganesyan, V.: Phenomenology of fully many-body-localized systems. *Phys. Rev. B* **90**(17), 174202 (2014)
3. Imbrie, J.Z.: On many-body localization for quantum spin chains. *J. Stat. Phys.* **163**(5), 998–1048 (2016)
4. Abanin, D.A., Altman, E., Bloch, I., Serbyn, M.: Colloquium: many-body localization, thermalization, and entanglement. *Rev. Mod. Phys.* **91**(2), 021001 (2019)
5. Schreiber, M.: Observation of many-body localization of interacting fermions in a Quasirandom optical lattice. *Science* **349**(6250), 842–845 (2015)
6. Smith, J., et al.: Many-body localization in a quantum simulator with programmable random disorder. *Nat. Phys.* **12**(10), 907–911 (2016)
7. Choi, J.-Y., et al.: Exploring the many-body localization transition in two dimensions. *Science* **352**(6293), 1547–1552 (2016)
8. Bordia, P., et al.: Probing slow relaxation and many-body localization in two-dimensional quasiperiodic systems. *Phys. Rev. X* **7**(4), 041047 (2017)
9. Roushan, P., et al.: Spectroscopic signatures of localization with interacting photons in superconducting qubits. *Science* **358**(6367), 1175–1179 (2017)
10. Kai, X., et al.: Emulating many-body localization with a superconducting quantum processor. *Phys. Rev. Lett.* **120**(5), 050507 (2018)
11. Braunstein, S.L., Caves, C.M.: Statistical distance and the geometry of quantum states. *Phys. Rev. Lett.* **72**(22), 3439 (1994)
12. Häffner, H., et al.: Scalable multiparticle entanglement of trapped ions. *Nature* **438**(7068), 643–646 (2005)
13. Wu, Y.-L., Das Sarma, S.: Understanding analog quantum simulation dynamics in coupled ion-trap qubits. *Phys. Rev. A* **93**(2), 022332 (2016)
14. Hauke, P., Heyl, M.: Many-body localization and quantum ergodicity in disordered long-range Ising models. *Phys. Rev. B* **92**(13), 134204 (2015)

15. Deng, D.-L., Li, X., Pixley, J.H., Wu, Y.-L., Das Sarma, S.: Logarithmic entanglement lightcone in many-body localized systems. *Phys. Rev. B* **95**(2), 024202 (2017)
16. De Tomasi, G., Bera, S., Bardarson, J.H., Pollmann, F.: Quantum mutual information as a probe for many-body localization. *Phys. Rev. Lett.* **118**(1), 016804 (2017)
17. Bañuls, M.C., Yao, N.Y., Choi, S., Lukin, M.D., Ignacio Cirac, J.: Dynamics of quantum information in many-body localized systems. *Phys. Rev. B* **96**(17), 174201 (2017)
18. Pal, A., Huse, D.A.: Many-body localization phase transition. *Phys. Rev. B* **82**(17), 174411 (2010)
19. Luitz, D.J., Laflorencie, N., Alet, F.: Many-body localization edge in the random-field Heisenberg chain. *Phys. Rev. B* **91**(8), 081103 (2015)
20. Robert Johansson, J., Nation, P.D., Nori, F.: QuTiP: an open-source python framework for the dynamics of open quantum systems. *Comput. Phys. Commun.* **183**(8), 1760–1772 (2012)
21. Brown, P.N., Byrne, G.D., Hindmarsh, A.C.: VODE: a variable-coefficient ODE solver. *SIAM J. Sci. Stat. Comput.* **10**(5), 1038–1051 (1989)
22. Bekenstein, J.D.: Black holes and entropy. In: Jacob Bekenstein: The Conservative Revolutionary, pp. 307–320. World Scientific (2020)
23. Lashkari, N., Stanford, D., Hastings, M., Osborne, T., Hayden, P.: Towards the fast scrambling conjecture. *J. High Energy Phys.* **2013**(4), 1–33 (2013). [https://doi.org/10.1007/JHEP04\(2013\)022](https://doi.org/10.1007/JHEP04(2013)022)
24. Landsman, K.A.: Verified quantum information scrambling. *Nature* **567**(7746), 61–65 (2019)
25. Hosur, P., Qi, X.-L., Roberts, D.A., Yoshida, B.: Chaos in quantum channels. *J. High Energy Phys.* **2016**(2), 1–49 (2016). [https://doi.org/10.1007/JHEP02\(2016\)004](https://doi.org/10.1007/JHEP02(2016)004)
26. Khemani, V., Lim, S.-P., Sheng, D.N., Huse, D.A.: Critical properties of the many-body localization transition. *Phys. Rev. X* **7**(2), 021013 (2017)
27. Nico-Katz, A., Bayat, A., Bose, S.: Information-theoretic memory scaling in the many-body localization transition. arXiv preprint [arXiv:2009.04470](https://arxiv.org/abs/2009.04470) (2020)




Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





On the Difference Between Shared Memory and Shared Address Space in HPC Communication

Atsushi Hori¹, Kaiming Ouyang², Balazs Gerofi^{1,3},
and Yutaka Ishikawa¹

¹ National Institute of Informatics, Tokyo, Japan
{ahori,yutaka_ishikawa}@nii.ac.jp

² University of California, Riverside, USA
kouya001@ucr.edu

³ RIKEN Center for Computational Science, Kobe, Japan
bgerofi@riken.jp

Abstract. Shared memory mechanisms, e.g., POSIX shmem or XPMEM, are widely used to implement efficient intra-node communication among processes running on the same node. While POSIX shmem allows other processes to access only newly allocated memory, XPMEM allows accessing any existing data and thus enables more efficient communication because the send buffer content can directly be copied to the receive buffer. Recently, the shared address space model has been proposed, where processes on the same node are mapped into the same address space at the time of process creation, allowing processes to access any data in the shared address space. Process-in-Process (PiP) is an implementation of such mechanism. The functionalities of shared memory mechanisms and the shared address space model look very similar – both allow accessing the data of other processes –, however, the shared address space model includes the shared memory model. Their internal mechanisms are also notably different. This paper clarifies the differences between the shared memory and the shared address space models, both qualitatively and quantitatively. This paper is not to showcase applications of the shared address space model, but through minimal modifications to an existing MPI implementation it highlights the basic differences between the two models. The following four MPI configurations are evaluated and compared; 1) POSIX Shmem, 2) XPMEM, 3) PiP-Shmem, where intra-node communication is implemented to utilize POSIX shmem but MPI processes share the same address space, and 4) PiP-XPMEM, where XPMEM functions are implemented by the PiP library (without the need for linking to XPMEM library). Evaluation is done using the Intel MPI benchmark suite and six HPC benchmarks (HPCCG, miniGhost, LULESH2.0, miniMD, miniAMR and mpiGraph). Most notably, mpiGraph performance of PiP-XPMEM outperforms the XPMEM implementation by almost 1.5x. The performance numbers of HPCCG, miniGhost, miniMD, LULESH2.0 running with PiP-Shmem and PiP-XPMEM are comparable with those of POSIX Shmem and XPMEM. PiP is not only a practical implementation of the shared

address space model, but it also provides opportunities for developing new optimization techniques, which the paper further elaborates on.

Keywords: Shared memory · Shared address space · Parallel execution model · HPC Communication · MPI

1 Introduction

Shared memory, which enables processes to access the same physical memory regions, is popular for implementing intra-node communication. The most well-known shared memory implementation is POSIX `shm` [13] that requires the allocation of a new memory region so that other processes can gain access to it. When implementing MPI inter-process communication, POSIX `shm` requires two memory copies; one copy from the send buffer to a shared memory region and another copy from the shared memory region to the receive buffer. XPMEM [9], which was first introduced on SGI supercomputers, enables other processes accessing existing memory regions. In XPMEM, the owner process of a memory region first *exposes* the memory region and then the process trying to access the memory region *attaches* the exposed region. If the memory region in which the send buffer resides is accessible from the receiver process, then only one memory copy enables the data transfer from the send buffer to the receive buffer. Thus, XPMEM is used in many MPI implementations to make intra-node communication more efficient than that of POSIX `shm`. Additionally, the XPMEM communication style also enables efficient implementation of one-sided communication. Once the addresses of the origin and target buffers are known, one-sided communications can take place without any intervention of the other process. XPMEM's big disadvantage, however, is the overhead of the attach operation. Consequently, most MPI implementations utilizing XPMEM make use of an XPMEM cache that reduces this overhead by caching attached memory regions [7].

The shared address space is another mechanism that enables processes to efficiently share memory. As of today, SMARTMAP [5], PVAS [19], MPC [18] and Process-in-Process (PiP) [10] implement this model. As the name suggests, in this model processes are created to share the same address space. Once processes are created, they can access any data in the address space and there is no need for any special operations that are required in the shared memory model, such as the creation of shared memory regions in POSIX `shm` or the `expose` and `attach` operations of XPMEM. Moreover, the shared address space model can provide all functionalities of the shared memory model, but with higher efficiency. There are more benefits to the shared address space. The address of an object is unique and does not depend on the accessing process. This means that complex data structures such as linked lists and/or tree structures where objects are linked with pointers can be also accessible without any extra effort. Even execution code can be shared. This characteristic of the shared address space model is very difficult to implement with traditional shared memory since that

does not guarantee the same virtual address for mapped objects. The shared address space, however, has also a disadvantage compared to the traditional shared memory. In particular, it may induce higher overhead when modifying the shared address space itself, such as by calling `m(un)map()` or `brk()` system calls that have no impact when using traditional shared memory.

The shared memory and the shared address space models can be confusing because both allow access to the data of another process. However, the shared memory model only allows access to some parts in the address space of the other processes, while the shared address space model allows for all processes to access everything in the same address space. The primary motivation of this paper is to clarify the difference between these two models.

The shared address space model can be defined as execution entities share an entire address space, not only parts of it. For example, the well-known multi-thread execution model can also be thought of as the shared address space model. However, the execution models of multi-process and multi-thread are very different, and it is not appropriate to compare the basic differences of the two memory models.

This paper highlights the differences between the two memory models both qualitatively and quantitatively. The possible optimization techniques enabled by using the shared address space are not the main concern of this paper, but some of them will be discussed in Sect. 6.

In this paper, we chose PiP as an implementation of the shared address space model because PiP is implemented at user-level and PiP can provide the XPMEM API. We chose MPICH for evaluation purposes as it is a well-known MPI implementation. We modified MPICH to create the shared address space environment, but the modification is minimized so that basic performance differences can be demonstrated. Our approach is to compare four different MPICH configurations by using POSIX `shmexec`, XPMEM, and PiP;

Shmem: configured to utilize POSIX `shmexec` for intra-node communication,
PiP-Shmem: configured to utilize POSIX `shmexec` but MPI processes are spawned by using PiP,
XPMEM: configured to utilize XPMEM for intra-node communication, and
PiP-XPMEM: configured to utilize XPMEM but XPMEM functions are implemented by using PiP.

The quantitative differences will be demonstrated by using the Intel MPI benchmark programs (IMB) and various HPC benchmarks. In summary, the main findings of this paper are as follows.

- PiP-Shmem behaves almost the same with Shmem,
- PiP-XPMEM outperforms XPMEM at best when the communication pattern is irregular and thus the XPMEM cache is ineffective,
- PiP(-XPMEM) has large overhead when calling the `m(un)map()` or `brk()` system calls, and
- application behavior does not trigger the above overhead and there are cases where PiP-Shmem and PiP-XPMEM outperforms Shmem and XPMEM.

To the best of our knowledge, this is the first paper which clarifies the differences between the shared memory and the shared address space models.

2 Background and Related Work

There are two well-used parallel execution models, multi-process and multi-thread. Although the MPI standard does not define which execution model is to be used to implement MPI, many MPI implementations are based on the multi-process model. OpenMP is a parallel language utilizing the multi-thread execution model. Usually, a process has its own address space, and nothing is shared among processes. Threads share almost everything including static variables. One may realize a third execution model to take the best of two worlds of multi-process and multi-thread. In this execution model, execution entities (processes or threads) share an entire address space, but each execution entity can run independently from the others, i.e., all variables are privatized unlike the thread model. Since an address space is shared, one can access the data owned by others whenever it's required. Specifically, nothing is shared in the process model, everything is shared in the thread model, and in this third execution model everything is shareable. Needless to say, the third execution model also provides the shared address space model. The term *process* may not be appropriate here because a *process*, as opposed to a thread, usually implies its associated address space. Hereinafter the term *task* is used in the contexts of the shared address space model and the third execution model.

The shared memory model does not have this feature since the mapped address of a memory segment may differ process by process. This feature is called Consistent Address View (CAV) in this paper. CAV enables the sharing of complex structures (i.e., linked list or tree) and execution codes. In most cases, those linked lists and tree structures hold pointers to refer to the other related object(s) and those related objects may widely scatter in an address space and may not fit in a memory segment. With traditional shared memory, pointers in complex data structures cannot be dereferenced as they are, and complex structures may not fit in a memory segment to share. Thus, sharing a complex structure with shared memory is difficult.

There are currently four major implementations of the third execution model; 1) SMARTMAP, 2) PVAS, 3) MPC, and 4) PiP. These are summarized in Table 1, *Base* indicates the base of implementation (process or thread), *Partition* indicates address space is regularly partitioned or not, *CAV* indicates if the implementation has the CAV feature, *Multi-Prog.* indicates if multi-program is supported or not, *PIE* indicates if executable must be PIE (Position Independent Code) or not, and *Impl. Lv.* indicates the implementation level, user or kernel.

The implementation of SMARTMAP relies on the page table structure of the x86 architecture and its page table has a unique format. A task is mapped twice in the shared address space, one for execution itself and another for accessing from the other tasks. Thus, to access the data of the other task an offset must be added to addresses and thus CAV is not supported by SMARTMAP.

Table 1. Shared Address Space Implementations

Name	Base	Partition	CAV	Multi-Prog.	PIE	Impl. Lv.	Note
PiP	Process	No	Yes	Yes	Yes	User	
SMARTMAP	Process	Yes	No	Yes	No	Kernel	Dedicated OS
PVAS	Process	Yes	Yes	Yes	Yes	Kernel	Patched Linux
MPC	Thread	No	Yes	No	No	User	Dedicated Compiler

A shared address space is partitioned in SMARTMAP and PVAS. All memory segments of a task are packed into one of the partitions. Unlike SMARTMAP, PVAS is architecture independent, and it loads an executable image onto one of the unused partitions. If a program A is loaded twice or more, the images of A must be loaded at different partitions. To enable this, the executable must be a PIE. This situation of PVAS is the same with PiP, while PiP does not partition an address space.

MPC has a different approach from the others. Its implementation is based on Pthread and MPC makes threads like processes. The variable privatization is implemented by converting static variables to Thread Local Storage (TLS) variables. This translation is done by a dedicated compiler and linker. The biggest issue with this implementation is that user programs may create (OpenMP) threads and declare their own TLS variables. The converted TLS variables must be able to be accessed by any (process-like) threads, while the user-declared TLS variables must be accessed only by the thread created by the user. So, the converted TLS variables and user-declared TLS variables have different accessing scopes. To solve this issue, they implement two different TLS systems; one for the converted TLS and another for user-defined threads. Despite this mitigation, MPC tasks still have the limitations coming from the thread implementation, such as only a single program can be loaded that shares a single file descriptor table, etc.

In the XPMEM shared memory model, the procedure to access the address space of another process is; 1) the exposing process must call `xpmem_make()` in advance, 2) then the accessing process calls `xpmem_get()`, and 3) calls `xpmem_attach()`. The `xpmem_make()` function of XPMEM is to specify an address range of the caller process so that the other process can attach only the memory regions within this specified address range. The `xpmem_get()` function is to check if the calling process can access the exposed memory. Finally, the `xpmem_attach()` function must be called to specify the memory region to be shared. The times to call XPMEM functions (and the times to call POSIX shm functions as well) are already reported by [10], showing `xpmem_get()` (and POSIX shm) overhead is very high.

Hashmi investigated various optimization techniques for implementing MPI by using XPMEM [7], though the title of his thesis has the term “Shared-Address-Space.” He proposed the XPMEM implementation of MVAPICH (an MPI implementation [20]) to improve P2P communications and collective operations. XPMEM cache was proposed to mitigate the high overhead. He also proposed optimization for handling MPI datatypes by using XPMEM.

SMARTMAP, PVAS and XPMEM are implemented at the kernel level. Consequently, it is very difficult to set up their environment on systems in operation if those systems do not support them already. To the contrary, PiP and MPC are implemented at user level, and it is easy to run programs under these environments on any system. As shown in Table 1, PiP is the most practical in terms of its transparency for a dedicated OS kernel, language processing system and CAV. This is the reason why we chose PiP in this paper.

The shared address space model is new and not thoroughly investigated yet. This paper is to clarify the basic characteristics of the shared address space model by comparing it with traditional shared memory. Possible applications of this model will be discussed in Sect. 6.

3 Process-in-Process (PiP)

In PiP, a normal process can spawn child PiP *tasks* located in the same address space of the spawning process. The parent process is called the *PiP root process* and the spawned tasks are called *PiP tasks*. When implementing MPI with PiP, the process manager process is the PiP root and MPI processes are PiP tasks. The PiP root process is also treated as a PiP task.

The PiP implementation relies on the `dlopen()` (not `dlopen()`) Glibc function and the Linux `clone()` system call. `dlopen()` loads a program with a new name space. By using this function, programs can be loaded into the same address space twice or more whilst maintaining their variable privatization. To load the same program but to a different location in the same address space, the loaded program must be compiled and linked as Position Independent Executable (PIE). The `clone()` system call is used to create a task sharing the same address space but to behave like a process, i.e., to have its independent file descriptor table, signal handlers, etc. Thus, a PiP task behaves just like a normal process except for the shared address space.

Let us explain about the variable privatization of PiP in a concrete example. Assume that a program has a statically allocated variable x and it spawns two PiP tasks derived from this program. Each PiP task has its own variable x at different location in the same virtual address space. Each task accesses its own variable x . Thus, the spawned tasks can run independently without having any collisions of accessing variables. This behavior is substantially different from that of the multi-thread mode where all static variables are shared. To access the variable x owned by another task, PiP provides several ways to pass the address of an object to another PiP task. If the address of an object to be accessed is known, then a task can simply access it without performing any extra operations.

This variable privatization makes PiP tasks much easier for programs to share an address space than that of using the multi-thread model. When a sequential program is run with the multi-thread model, static variables must be protected from simultaneous access. In the shared address model, however, all static variables are privatized and there is no need for such protection. Thus, multiple instances of a program or multiple programs can run in the shared address space without the need for any modification of their source code.

There is one big limitation of PiP that comes from the current Glibc implementation. The number of name spaces which the `dlopen()` can create is limited up to 16. This number is the size of a statically allocated array of name spaces and it is hard coded in Glibc. This number is too small considering the number of CPU cores on a node. As a result, we patched Glibc so that more than 16 PiP tasks can be created. Regardless of using the patched Glibc or not, the current PiP implementation is purely user-level, and requires neither any kernel patch nor a specific kernel module. It should be noted that this patched Glibc can coexist with the existing Glibc. Only when a PiP program is compiled and linked by using the PiP compiler wrapper scripts, the patched Glibc is used.

The PiP library also provides the XPMEM functions and the XPMEM header file so that programs using XPMEM can easily be converted to PiP-aware programs. Furthermore, a converted program can run without installing the XPMEM kernel module. The XPMEM functions implemented in the PiP library do nothing and work very efficiently because PiP tasks can access any data in the address space to begin with.

4 Shared Memory Vs. Shared Address Space

4.1 Page Tables and Page Faults

In modern CPUs and OS kernels, an address space is essentially implemented as a page table located inside the kernel. The page table holds all mapping information from virtual addresses to physical addresses on every memory page in use. Every process has its own address space (left figure of Fig. 1). This implies every process has its own page tables. When a shared region is created by using POSIX `shmem`, the physical memory pages that are share are mapped in the page tables associated with the processes to share the memory region. In XPMEM, an existing memory region to share must be exposed and then the region is attached by the other process.

Let us take a closer look at the creation of new mappings. In many modern OSes including Linux, there are two steps; one to create a skeleton of the mapping upon request, followed by a (minor) page fault when accessing the memory page for the first time. This page fault triggers the creation of a page table entry of the memory page. Every step is accompanied with non-negligible overhead. Furthermore, in the shared memory model, these steps happen on every process accessing the shared memory region. Thus, the shared memory model may suffer from the setup overhead and the overhead of a large number of page faults.

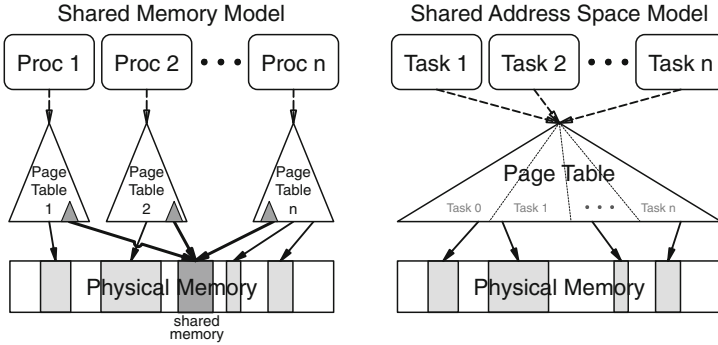


Fig. 1. Page table structure difference

To the contrary, there is only one page table regardless of the number of tasks in the shared address space model (right figure of Fig. 1). Once a page table entry is created, the corresponding memory page can be accessed by any tasks sharing the address space without triggering page faults. Nowadays the number of tasks (processes) in a node for parallel execution can be large since the number of CPU cores is increasing. Thus, the overhead of page table setup and page faults can be far less than that of the shared memory model.

4.2 Modifications to Page Tables

However, the shared address space has also a disadvantage. Suppose that there are four processes sharing the same address space (right figure of Fig. 1). In theory, the size of the shared page table is the sum of the sizes of the four independent processes. The bigger page table than that of shared memory can take longer time to walk through the page table. Additionally, the page table is shared by the four processes. To maintain consistency of the page table, the page table must be protected from the simultaneous modification by using some combination of locking. This locking renders the overhead of page table modifications even larger.

There are two well-used system calls, `m(un)map()` and `brk()`, to modify the page table when memory regions get (de)allocated. The `mmap()` system call is also used to allocate a shared memory region. The `brk()` system call extends the memory region of the heap segment. The `brk()` function is used by the `malloc()` routines. The detailed `m(un)map()` overhead was already analyzed and reported by the original PiP paper [10], and the overhead on PiP is almost the same with that of Pthreads which is another implementation of the shared address space model.

Table 2. Shared memory and shared address space

Feature	Shared memory	Shared address space (PiP)
Sharing target	Regions	Entire address space
Setup cost	Very high	Nothing
Page table modification cost	–	Higher
# Page Faults	Larger	Smaller
Lifespan of sharing	Attach \sim detach	Same as task/thread
Consistent Address View (CAV)	No	Yes

Summary of the Differences

Table 2 shows the summary of a qualitative comparison between the shared memory model and the shared address space model.

5 Evaluation

The objective of this evaluation is to assess whether or not the shared address space can provide performance advantages compared to shared memory in the presence of the advantages and disadvantages described in the previous section. Unfortunately, the benchmark programs evaluated in this paper have no usage of CAV which is one of the most unique features of shared address space, which we will further discuss in Sect. 6.

We chose MPICH (Version 3.4.1) for evaluation. Although various optimizations based on PiP are possible, we kept modifications to MPICH minimal to highlight the basic difference between the shared memory and shared address space models. To make MPICH PiP-aware, we modified the Hydra process manager of MPICH to spawn PiP tasks instead of creating normal processes. MPICH was configured in four ways,

Shmem MPICH is configured to use POSIX shmem for intra-node communication,

XPMEM MPICH is configured to use XPMEM for intra-node communication, if possible,

PiP-Shmem MPICH is configured to spawn PiP tasks and PiP tasks allocate POSIX shared memory regions for intra-node communication (although shmem is not needed with PiP), and

PiP-XPMEM in addition to PiP-Shmem, XPMEM code is enabled but implemented by the PiP library and the XPMEM cache code is bypassed.

A certain difference between XPMEM and PiP-XPMEM is expected because XPMEM incurs the overhead of attaching memory of other processes as well as the overhead of the XPMEM cache to reduce the number of calls to XPMEM attach. The performance of PiP-Shmem and PiP-XPMEM, however, might incur higher `mmap()` and/or `brk()` overhead.

P2P and RMA performances were measured by using Intel MPI Benchmark [11]. Six mini-apps, HPCCG [8], miniGhost [3], LULESH2.0 [1, 12], miniMD [14], miniAMR [2] and mpiGraph [15], were chosen to cover various parallel execution and communication patterns.

We confirm that XPMEM is used in MPICH when calling P2P functions (Send/Recv, Isend/Irecv and Sendrecv) with message sizes larger than or equal to 4KiB and some RMA calls (Get/Put and Accumulate). This condition is the same with PiP-XPMEM since the threshold setting was left unchanged.

5.1 Experimental Environment

To measure the four MPICH configurations, Shmem, PiP-Shmem, XPMEM and PiP-XPMEM, we needed access to a cluster where XPMEM was already installed. Unfortunately, only a limited number of compute nodes from the Oakforest-PACS supercomputer [12] could be installed with such environment. Table 3 describes our evaluation environment.

Table 3. Experimental platform information

H/W	CPU	# Cores (# HT)	Clock	Memory	Network
	Xeon Phi 7250 (KNL)	68 (272)	1.4 GHz	96(+16) GiB	Omni-Path
S/W	OS	Linux 3.10.0-957.27.2.el7.x86_64			
	PiP-Glibc	2.17-260.el7.pip.branch			
	GCC	4.8.5 20150623 (Red Hat 4.8.5-36)			
	Intel MPI	Intel(R) MPI Library 2019 Update 5 for Linux			

In all cases in this section, MPI processes are bound to CPU cores with the `-bind-to rr` (round-robin) MPICH runtime option. No other runtime option is specified. The performance numbers of the benchmark programs compiled and linked with built-in Intel compiler and Intel MPI will also be shown in some cases, just for reference. All MPICH libraries and mini applications are compiled using GCC. All measurements were repeated ten times and average numbers are reported.

5.2 Intel MPI Benchmark (IMB) Performance

To measure and compare P2P performance in this subsection, all benchmark numbers in `IMB-MPI1` and `IMB-RMA` were measured using only a single node. Most benchmark results did not show any big difference between Shmem and PiP-Shmem and between XPMEM and PiP-XPMEM, respectively. Here, Exchange in `IMB-MPI1` (Fig. 2) and `All_put_all` in `IMB-RMA` (Fig. 3) results are shown. In the Exchange benchmark, MPI processes form a ring topology and each MPI process send messages to its neighbors by calling `MPI_Isend()`, `MPI_Irecv()`

and `MPI.Wait()`. In the `All_put_all` benchmark, each MPI process puts data to all the other MPI processes. Remember that XPMEM is only effective in the P2P communication and when the message size is larger than or equal to 4KiB, and some RMA operations including `get` and `put`.

In Fig. 2, Shmem, not shown in this figure because it is the base (always one), and PiP-Shmem exhibited the very similar latency curves, except for the dip at message size of 64 KiB. This is because the latency of Shmem is exceptionally large at that message size. XPMEM and PiP-XPMEM exhibit almost the same and much better than that of Shmem and PiP-Shmem when the message size is larger than or equal to 4 KiB which is the threshold to call the XPMEM functions to communicate.

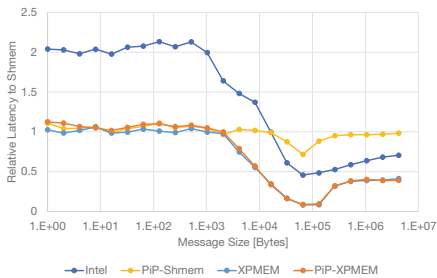


Fig. 2. IMB-MPI1 Exchange (-np 32 -ppn 32)

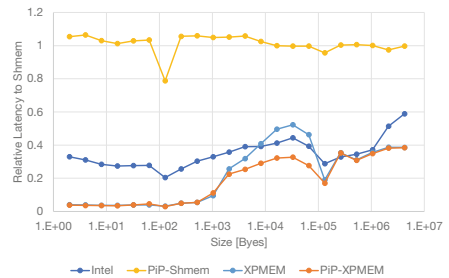


Fig. 3. IMB-RMA All_put_all (-np 32 -ppn 32)

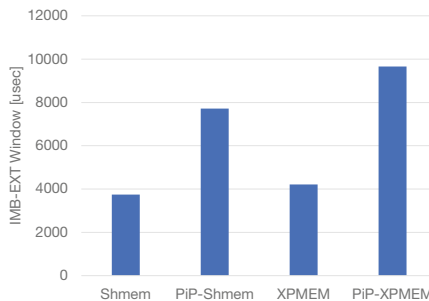


Fig. 4. IMB-EXT Window (-np 32 -ppn 32)

In Fig. 3, Shmem and PiP-Shmem exhibited almost the same except for the dip at 1KiB. Again, this dip comes from the exceptional Shmem latency at this message size and affects the ratios of the other MPI configurations. Comparing XPMEM and PiP-XPMEM, PiP-XPMEM exhibited much better latency at the range from 4KiB to 64KiB. We believe this performance advantage of

PiP-XPMM over XPMM comes from the XPMM cache misses. It has been shown by Hashmi that the XPMM cache miss overhead can only be seen on smaller message sizes [7] and Fig. 3 matches with his report.

There is a pitfall in the IMB-RMA benchmark. The measured time does not include the time to create an RMA window. The time to call `MPI_Window_create()` can be measured by another program IMB-EXT in the IMB suite. Figure 4 shows the results using this program. By large, PiP-Shmem and PiP-XPMM took more than 2x compared with Shmem and XPMM, respectively. This PiP overhead is considered to come from the contention of `mmap()` calls and the larger size of the page table. In general, the RMA window creation function is called at the initialization stage of an MPI program, and the RMA window creation function is not called frequently. So this overhead is diluted in real applications.

5.3 Mini App Performance

Table 4 lists the mini applications used in this subsection and parameters to run them. The column *Maj. MPI send* indicates the MPI function which is called most frequently in that application. The *Perf. Index* at the last column

Table 4. Benchmark parameters

Name (Lang.)	-np	Parameters	Maj. MPI Send	Perf. Index
HPCCG (C++)	32	150 150 50	Send	MFLOPS
	256	(same as above)		
miniGhost (F90)	32	ndim = 600 npx = ppy = 4 ppz = 2 scaling = 1	Isend	Total GFLOPS
	256	ndim=1200 npx=ppy=8 ppz=4 scaling=1		
LULESH2.0 (C++)	27	s = 16	Isend	FOM
	125	s = 18		
miniMD (C++)	32	in.lj.miniMD	Sendrecv	Performance
	256	(same as above)		
miniAMR (C)	32	npx = npy = 4 npz = 2 max_blocks = 1000 nx = ny = nz = 8 init_x = init_y = init_z = 1 num_refine = 4 num_objects = 1 object = 2 0 -0.01 -0.01 -0.01 0.0 0.0 0.0 0.0 0.0 0.0 0.0009 0.0009 0.0009 num_tsteps = 200 comm_vars = 2	Isend	Total GFLOPS
	256	npx = npy = 8 npz = 4 (rests are same as above)		
mpiGraph (C)	32	1048576 10 10	Isend	Send avg
	256	(same as above)		

indicates which number reported by the application is used for the performance comparison.

In the following application evaluation, single-node performance and multiple-node performance are shown and compared. The number of MPI processes of LULESH2.0 must be cubic, so the number of MPI processes of LULESH2.0 evaluation is 27 for single-node, 125 for multiple nodes (five nodes). The number of MPI processes of all the other applications is 32 for running on a single node and 256 for running on eight nodes.

Figure 5 shows the single node performance ratios of mini apps based on Shmem performance. As seen, the PiP-Shmem, XPMEM and PiP-XPMEM performs within the range of few percent differences. Most notably, the performance of mpiGraph running with PiP-XPMEM outperforms Shmem at 3x, XPMEM at 1.5x.

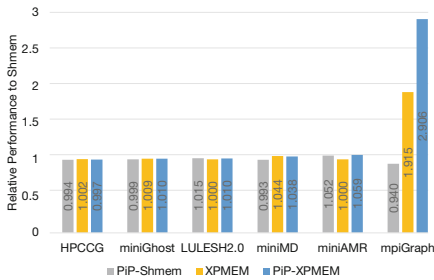


Fig. 5. Application Performance Comparison

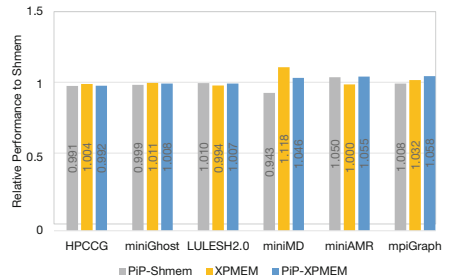


Fig. 6. Application Performance Comparison (multiple nodes)

Figure 6 shows the multi-node performance numbers. Unlike Fig 5, the big performance gain of PiP-XPMEM on mpiGraph is hardly seen. Further, XPMEM outperforms PiP-XPMEM and Shmem also outperforms PiP-Shmem in miniMD. In mimiAMR, PiP-Shmem and PiP-XPMEM outperform Shmem and XPMEM by about 5%, respectively.

In the next subsection, we will try to analyze these situations in terms of the XPMEM cache miss, the number of page faults, the number of brk() calls, and the communication patterns.

Detailed Analysis

Figure 9 shows the numbers of XPMEM cache accesses and the XPMEM cache miss ratios on each application. The XPMEM cache works in such a way that firstly it searches the XPMEM cache table and if there is no cache entry then the xpmem_get() and xpmem_attach() are called to attach the memory region and the attached region is registered in the XPMEM cache. The upper bars in the figure indicate the number of XPMEM cache access and the lower bars indicate the cache miss ratios. In the mpiGraph application, the XPMEM cache miss ratio is exactly 10%, the highest among the others.

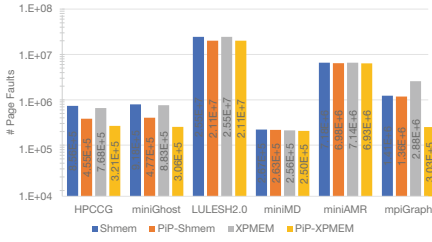


Fig. 7. # Page Faults (single node)

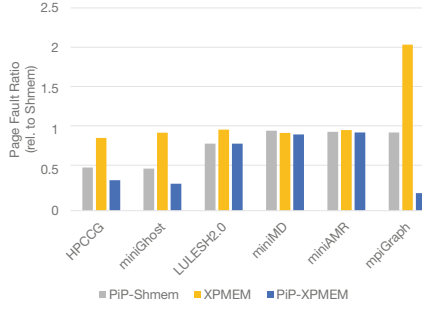


Fig. 8. Page Fault Ratio (single node)

Figure 7 shows the number of page faults and Fig. 8 shows the ratio of page faults during the executions. The numbers of page faults with PiP-Shmem and PiP-XPMEM are always less than those of Shmem and XPMEM respectively, as expected. Most notably, the number of page faults of XPMEM on mpiGraph is 2x higher than that of Shmem and close to 10x higher than that of PiP-XPMEM.

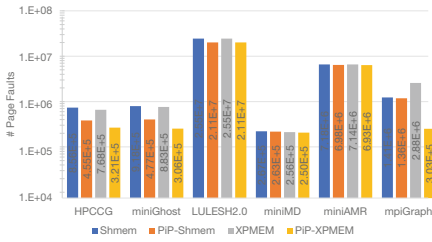


Fig. 9. # XPMEM Cache Access and Cache Miss Ratio (single node)

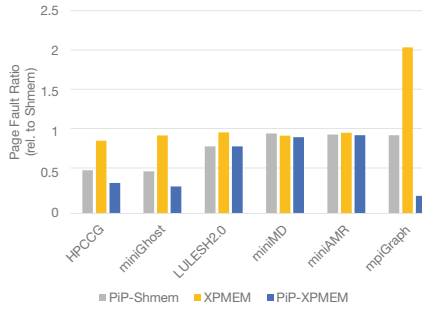


Fig. 10. Number of brk() calls

The number of `mmap()` system call includes the calls when loading a program and required shared libraries and the comparing the number of `mmap()` calls may be imprecise. Instead, the number of `brk()` system calls is measured in this paper. We measured the overhead of the `brk()` system call on 32 tasks on a single node. The overhead of the PiP-Shmem case is very high, almost 200x of the Shmem case. This large overhead may affects the application performance.

Figure 10 shows the numbers of `brk()` calls per node. These numbers are measured by using the Linux `strace` command. The `brk()` is always called in pairs, one to obtain the current heap address and another to increase the heap segment. So, the actual number of page table modifications is the half of the numbers shown in the table. Unfortunately, the `strace` command is not PIE and

it is impossible to run it on the PiP environment. However, the numbers would be the same with the numbers in the graph, since the MPICH code modifications to be PiP-aware do not affect the number of `brk()` calls. These numbers are almost independent from single node or multiple nodes, and using XPMEM or not, except for LULESH2.0 with which the number of page faults on multiple nodes are higher than those on a single node.

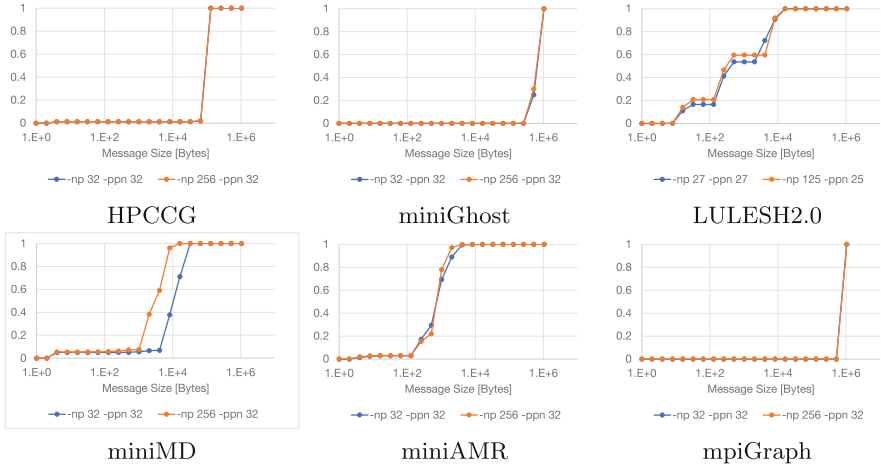


Fig. 11. Communication patterns

Figure 11 shows the cumulative graphs of send frequency over message sizes. These numbers are obtained by using the PMPI interface. The communication pattern of miniMD depends on the number of ranks while the others are almost independent from the number of nodes. In miniMD, the larger the number of ranks, the smaller the message sizes. This is due to the fact that the miniMD parameter setting is in a strong scaling way.

Let us examine all these evaluation results. The single node performance of mpiGraph is a good showcase of how PiP-XPMEM works better than XPMEM; 1) high XPMEM cache miss ratio, 2) high number of page faults, 3) moderate number of `brk()` calls, and 4) exchanging large messages by which MPICH can utilize the XPMEM functionalities.

LULESH2.0 and miniAMR exhibited similarly in terms of high number of XPMEM calls, low XPMEM cache miss ratio, and high number of `brk()` calls. The high number of `brk()` calls is considered to be the disadvantages for PiP-Shmem and PiP-XPMEM, however, there are almost no performance penalties observed. On the contrary, PiP-Shmem and PiP-XPMEM slightly outperforms them. This can be explained by the fact of the high number of page faults (Fig. 7).

HPCCG and miniGhost performed constantly independent from whether using PiP and/or XPMEM or not. The number of page faults could be reduced by

using PiP or PiP-XPMMEM. This advantage of PiP can be considered to be canceled by the `brk()` overhead. The advantage of using XPMMEM over Shmem on HPCCG, miniGhost and miniMD might have been spoiled by the 1% XPMMEM cache miss ratio (Fig. 9).

The miniMD multiple node performance of PiP-Shmem is slightly worse than that of Shmem. miniMD exhibited almost the same with HPCCG and miniGhost in terms of the number of page faults, XPMMEM cache miss ratio, and the number of the `brk()` calls. A big difference between miniMD and those applications can be found at the communication pattern (the miniMD graph in Fig. 11). The message sizes of running on 256 ranks are smaller than those of running on 32 ranks. The smaller the message size, the larger the impact of the `brk()` overhead.

6 Discussion

So far, the performance of PiP has been evaluated by using MPICH with the indispensable and minimal modifications to be PiP-aware. However, there is a lot of room for optimizing MPI implementations by using the shared address space model. In the POSIX Shmem usage of an MPI implementation, there are two `mmap()` calls; one to allocate a shared memory region and another to attach the shared memory region to access. In the shared address space model, however, only one call to allocate memory is enough. Once the memory region is allocated the memory region can be accessible without calling another `mmap()` call to attach. If an MPI implementation is optimized to utilize the full advantages of the shared address space, then the number of `mmap()` calls can be halved. Although the overhead of modifying a page table is high in shared address space, the smaller number of page table modifications may lead to smaller overhead.

The shared address space may improve not only intra-node communication performance, but also inter-node communication performance. Ouyang is eagerly working on MPI optimization to improve inter-node, not intra-node, communication by using PiP. In [17], Ouyang et al. proposed *CAB-MPI* where communication queue entries of the other MPI processes are stolen to balance communication load among MPI processes in a node. In their other paper [16], *Daps* is proposed so that idle MPI processes steal the asynchronous progress work of the other busy MPI processes, instead of creating an asynchronous progress thread. CAV provided by the shared address space model plays a very important role when implementing CAB-MPI and Daps. The message queue (i.e., send or receive queue) is implemented as a linked list in many cases. CAV enables CAB-MPI to access the message queues of the other processes without largely modifying the existing queue structures. To implement the asynchronous progress stealing in Daps, things are more complicated than implementing CAB-MPI. In MPICH, low level communication functions are called via function pointers to decouple the device independent code and device dependent code. Those functions must also be called by a different MPI process when implementing Daps, and the CAV nature which the shared address space model provides enables this.

Although this paper focused on the difference between the shared memory and the shared address space and reported advantages in intra-node MPI communication, there are many other potential applications which would have benefits by using the shared address space. The shared address space can also be applied to various communication libraries (i.e., OpenSHMEM [4]) and parallel programming languages (i.e., PGAS languages).

In-situ applications, visualization programs, and multi-physics applications are required to run two or more programs simultaneously and these programs cooperate with the others and exchange information among them. What if these programs run in the shared address space environment? The data exchange among programs can be more efficient than that of the conventional ways, – data exchange via file or coupling library – because data of the other program can be accessed directly. There are many open issues in this field of coupling multiple programs, however, it is our belief that the shared address space can be an answer for the question, how to connect programs in an efficient way.

Garg, Price and Cooperman proposed a checkpoint-restart system named *Mana* which is agnostic to MPI implementations and network devices by having a dedicated thread to save a memory image [6]. Although the authors claimed that their approach is hard to implement by using PiP, we think checkpoint-restart is a very attractive and challenging application of the shared address space model.

7 Summary

This paper has provided a detailed comparison between the shared memory and shared address space models. Although these two models appear similar since both models allow access to data owned by the other processes, their underlying mechanisms are notably different. From a qualitative point of view, the shared address space model may have fewer number of page table modifications and page faults than those of the shared memory model. To the contrary, the shared address space model may incur larger overhead when modifying page tables, e.g., when calling `m(un)map()` and `brk()` system calls. This overhead comes from the shared page table among processes and from the fact that the page table size is larger than that of the shared memory model. From a quantitative point of view, evaluations were conducted by using P2P benchmark programs and mini application benchmark programs. PiP is chosen as an implementation of the shared address model and an MPI implementation was modified in a minimal way to have the shared address space model. Four MPI configurations; 1) (POSIX) Shmem, 2) XPMEM, 3) PiP-Shmem and 4) PiP-XPMEM are compared. Shmem and XPMEM are as the representatives of the shared memory model, PiP-Shmem and PiP-XPMEM are the representatives of the shared address model.

The P2P benchmarks show that both models perform comparably. RMA benchmark reveals that RMA operation of the shared address space may outperform the shared memory model, however, the RMA window creation of the

shared address space model is almost twice as costly as that of the shared memory model. Most mini benchmark programs also perform comparably. Most notably, mpiGraph performance with PiP-XPMM outperformed Shmem by 3x and XPMM by 1.5x.

The shared memory model is an old technology which has received a lot of attention in the literature. To the contrary, the shared address space model is newer and only a few investigations have been done so far. We believe that considering the opportunities to improve the current HPC system software, it is worth investigating the shared address space model. This paper has made the first steps towards this direction.

PiP is an open-source software and freely available at <https://github.com/procinproc/procinproc.github.io>. The PiP package includes the patched Glibc, PiP-aware GDB, installation program (named `pip-pip`), and more. PiP can also be installed by using Spack (<https://github.com/spack>) with the package name of `process-in-process`.

Acknowledgment. This work has been partially funded by JST AIP Grant Number JPMJCR19U2. We thank the University of Tokyo, the University of Tsukuba, and JCAHPC for letting us access the OFP machine and for their help with getting the experiments done.

References

1. Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical report, LLNL-TR-490254, July 2011
2. miniamr, version 00 (2014). <https://www.osti.gov/servlets/purl/1253324>
3. Barrett, R.F., Heroux, M.A., Vaughan, C.T.: MiniGhost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing. <https://doi.org/10.2172/1039405>. <https://www.osti.gov/biblio/1039405>
4. Brightwell, R., Pedretti, K.: An intra-node implementation of OpenSHMEM using virtual address space mapping. In: Fifth Partitioned Global Address Space Conference. Galveston Island, Texas (2011). <http://pgas11.rice.edu/papers/BrightwellPedretti-OpenSHMEM-PGAS11.pdf>
5. Brightwell, R., Pedretti, K., Hudson, T.: SMARTMAP: operating system support for efficient data sharing among processes on a multi-core processor. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008, pp. 25:1–25:12. IEEE Press, Piscataway (2008). <http://dl.acm.org/citation.cfm?id=1413370.1413396>
6. Garg, R., Price, G., Cooperman, G.: Mana for MPI: MPI-agnostic network-agnostic transparent checkpointing. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2019, pp. 49–60. Association for Computing Machinery, New York (2019). <https://doi.org/10.1145/3307681.3325962>
7. Hashmi, J.M.: Designing high performance shared-address-space and adaptive communication middlewares for next-generation HPC systems. Ph.D. thesis, The Ohio State University (2020)
8. Heroux, M.A., Dongarra, J., Luszczek, P.: HPCG benchmark technical specification. <https://doi.org/10.2172/1113870>. <https://www.osti.gov/biblio/1113870>

9. Hjelm, N.: Linux Cross-Memory Attach. <https://github.com/hjelmn/xpmem>
10. Hori, A., et al.: Process-in-process: techniques for practical address-space sharing. In: Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2018, pp. 131–143. Association for Computing Machinery, New York (2018). <https://doi.org/10.1145/3208040.3208045>
11. Intel Corp.: Intel MPI Benchmarks - User Guide and Methodology Description, Revision 3.2.4 edn. (2013)
12. Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Technical report, LLNL-TR-641973, August 2013
13. Kerrisk, M.: shm_overview(7) - Linux manual page (2021). https://man7.org/linux/man-pages/man7/shm_overview.7.html
14. Li, M., Lin, J., Lu, X., Hamidouche, K., Tomko, K., Panda, D.K.: Scalable MiniMD design with hybrid MPI and OpenSHMEM. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014. Association for Computing Machinery, New York (2014). <https://doi.org/10.1145/2676870.2676893>
15. Moody, A.: mpigraph, version 00 (2007). <https://www.osti.gov/servlets/purl/1249421>
16. Ouyang, K., Si, M., Hori, A., Chen, Z., Balaji, P.: Daps: a dynamic asynchronous progress stealing model for MPI communication. In: 2021 IEEE International Conference on Cluster Computing (CLUSTER), pp. 516–527 (2021). <https://doi.org/10.1109/Cluster48925.2021.00027>
17. Ouyang, K., Si, M., Hori, A., Chen, Z., Balaji, P.: CAB-MPI: exploring interprocess work-stealing towards balanced MPI communication. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Press (2020)
18. Pérache, M., Carribault, P., Jourdren, H.: MPC-MPI: an MPI implementation reducing the overall memory consumption. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) EuroPVM/MPI 2009. LNCS, vol. 5759, pp. 94–103. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03770-2_16
19. Shimada, A., Gerofi, B., Hori, A., Ishikawa, Y.: Proposing a new task model towards many-core architecture. In: Proceedings of the First International Workshop on Many-Core Embedded Systems, MES 2013, pp. 45–48. ACM, New York (2013). <https://doi.org/10.1145/2489068.2489075>
20. The Ohio State University: MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. <http://mvapich.cse.ohio-state.edu>














Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Evaluating GPU Programming Models for the LUMI Supercomputer

George S. Markomanolis¹ , Aksel Alpay², Jeffrey Young⁵ ,
Michael Klemm³ , Nicholas Malaya³ , Aniello Esposito⁴ ,
Jussi Heikonen¹ , Sergei Bastrakov⁶, Alexander Debus⁶ , Thomas Kluge⁶ ,
Klaus Steiniger⁶ , Jan Stephan^{6,7}  , Rene Widera⁶ ,
and Michael Bussmann^{6,7} 

¹ CSC - IT Center for Science Ltd., Espoo, Finland

{georgios.markomanolis, jussi.heikonen}@csc.fi

² Heidelberg University, Heidelberg, Germany

aksel.alpay@uni-heidelberg.de

³ Advanced Micro Devices Inc, Santa Clara, USA

{michael.klemm, nicholas.malaya}@amd.com

⁴ Hewlett Packard Enterprise, Spring, USA

aniello.esposito@hpe.com

⁵ Georgia Institute of Technology, Atlanta, USA

jyoung9@gatech.edu

⁶ Center for Advanced Systems Understanding, Görlitz, Germany

⁷ Helmholtz-Zentrum Dresden-Rossendorf, Dresden, Germany

{s.bastrakov, a.debus, t.kluge, k.steiniger,

j.stephan, r.widera, m.bussmann}@hzdr.de

Abstract. It is common in the HPC community that the achieved performance with just CPUs is limited for many computational cases. The EuroHPC pre-exascale and the coming exascale systems are mainly focused on accelerators, and some of the largest upcoming supercomputers such as LUMI and Frontier will be powered by AMD Instinct™ accelerators. However, these new systems create many challenges for developers who are not familiar with the new ecosystem or with the required programming models that can be used to program for heterogeneous architectures. In this paper, we present some of the more well-known programming models to program for current and future GPU systems. We then measure the performance of each approach using a benchmark and a mini-app, test with various compilers, and tune the codes where necessary. Finally, we compare the performance, where possible, between the NVIDIA Volta (V100), Ampere (A100) GPUs, and the AMD MI100 GPU.

Keywords: GPU · Programming models · HIP · CUDA · OpenMP · hipSYCL · Kokkos · Alpaka

1 Introduction

Europe has procured a number of supercomputers through the EuroHPC Joint Undertaking (JU) organization. In this work, we focus on the LUMI [1] super-

computer which is being installed in Finland by Hewlett-Packard Enterprise and is run by a consortium of ten European countries. LUMI will have both a CPU and a GPU partition, where the CPU partition performance is only a few petaflops, the AMD Instinct™ GPUs provide almost 0.5 EFLOPS across 2560 nodes with 64 core AMD Trento CPU and four AMD MI250X GPUs, using similar technology as the Frontier system [2].

There are a few parallel older programming models, however, with the arrival of the GPUs, other programming models had to be created, such as Compute Unified Device Architecture (CUDA) [3], OpenCL [4], or directive-based programming models such as OpenMP [5] and OpenACC [6]. Meanwhile, even more programming models have emerged, some of which are more widely known than others. For some of them, there is a significant learning curve, and others are to be used by HPC domain experts.

When a scientist prepares an application to be ported to a GPU architecture, or to move from NVIDIA GPUs to AMD GPUs, the effort often depends on the used programming model. With such a wide variety of available programming models, it sometimes is not straightforward which one to use. In this paper, we explore the porting procedure for the LUMI supercomputer, discuss the applicable programming models, and present some results of various benchmarks and performance comparisons across AMD and NVIDIA GPUs.

The main contributions of this work are as follows:

- We present a porting diagram that illustrates how the LUMI users could port their application in various scenarios.
- To our knowledge, this is one of the first comparisons between NVIDIA V100/A100, and AMD MI100.
- We evaluate the performance of many programming models such as HIP, CUDA, OpenMP Offloading, hipSYCL, Kokkos, and Alpaka while optimizing when possible.
- We present results of the BabelStream with Alpaka backend for the first time.
- We present how to tune some of the kernels.

2 Related Work

For many programming models, there are studies that evaluate these for CPUs or GPUs. In [7] the authors study OpenMP offload on NVIDIA V100 with a few mini-apps and various compilers, observe performance variations, and provide some OpenMP optimization techniques. In [8], the authors present the compute-bound mini-app miniBUDE and evaluate various programming models, including offload to GPUs. In [9], the authors present a performance analysis of CUDA, OpenACC, and OpenMP programming models on V100 GPU where they illustrate how it is easier to use OpenMP offloading and OpenACC compared to CUDA, and they measure the performance. Deakin et al. in [10] evaluate the performance of benchmarks in SYCL and comparing them with an OpenCL version. They use three applications for this purpose. The authors in [11] present a performance portability study on different CPUs/GPUs, using programming

models and codes to investigate the performance portability. However, not many of them supported an AMD GPU at that time.

Compared to the current related work, we use some new GPUs, and especially the AMD MI100 and evaluate the programming models and tune them based on its hardware specifications. Furthermore, we try to use the most recent versions of the programming models where possible. Finally, we provide box plots in some cases to identify variations.

3 Programming Models

In this section, we present a few programming models that we plan to use on LUMI, and later we describe in which situation to use them.

3.1 HIP

The Radeon Open Compute (ROCm) platform [12,13] includes programming models to develop codes for AMD GPUs. Among those is the Heterogeneous-compute Interface for Portability (HIP) [14]. HIP is a C++ API and kernel language to create portable applications for the AMD ROCm platform as well as NVIDIA GPUs using the same source code. It is open source, it provides an API to port your code, and the syntax is very similar to CUDA. It supports a large subset of the CUDA runtime functionality and has almost no negative performance impact over coding directly in CUDA. HIP includes features such as C++11 lambdas, namespaces, classes, templates, etc. The HIPify [15] tools convert CUDA code to HIP. Of course, tuning will be required for each specific GPU.

Table 1 exemplifies some similarities between CUDA and HIP. For most cases, replacing the “cuda” in the function name with “hip” as well as for the arguments is enough to translate the API. However, not all the CUDA API is supported in HIP yet. Executing a GPU kernel is similar as you can see in the corresponding table but there is also a HIP API called `hipLaunchKernelGGL`.

With the HIP translation tool, a common approach is to semi-automatically port a CUDA code to HIP. For example, to convert a CUDA file called *example.cc*, the command `hipify-perl --inplace example.cc` performs the translation of CUDA API calls to HIP API calls (a backup of the original code is kept as `example.cc.hip`). There is also `hipconvertinplace-perl.sh` to translate all source files of a directory as well as a version of the HIPify tool that is based on the clang compiler. For more details about porting codes there are a few sources such as [16,17].

For CUDA Fortran codes, it is required to do the following steps (further details are available at [17]):

- Port CUDA Fortran code to HIP kernels in C++. The `hipfort` API helps to invoke the HIP API from Fortran.
- Wrap the kernel launch in function with C calling convention.
- Call the launch function from Fortran through the Fortran 2003 C bindings.

Table 1. Convert CUDA code to HIP

CUDA	HIP	Description
<code>cudaMemcpy</code>	<code>hipMemcpy</code>	Copy data between two different memory locations
<code>cudaMalloc</code>	<code>hipMalloc</code>	Allocates a memory pointer on the device
<code>cudaFree</code>	<code>hipFree</code>	Deallocate memory from the GPU
<code>kernel_name <<< gridsize, blocksize, shared_mem_size, stream >>>(arg0, arg1, ...);</code>	<code>kernel_name <<< gridsize, blocksize, shared_mem_size, stream >>>(arg0, arg1, ...);</code>	Execute a GPU kernel

3.2 The OpenMP Application Programming Interface

The OpenMP API supports offloading computation to accelerator devices since version 4.0 and has since then refined and extended the features continuously [18]. The OpenMP API supports a variety of *target* directives that control the transfer of data (if needed), transfer of control flow, as well as parallelism on the target device. OpenMP also offers low-level API interfaces for memory allocation and data transfers similar to the interfaces of the CUDA and HIP programming models.

This is a very basic example of an OpenMP offload region, running code on a GPU:

```
#pragma omp target teams distribute parallel for simd \
        map(to:A[:N]) map(from:B[:N]) \
        num_teams(x) thread_limit(y)
for (int i = 0; i < N; ++i) {
    B[i] = expression(A[i], i);
}
```

In the example above, the `target` construct transfers the control flow from the host device to the default target device (the host thread will await completion of the offload region). The `map` clauses are used to specify the data that is needed for execution as well as the direction of the data flow. If the host and accelerator have distinct memories, the OpenMP implementation will perform an actual transfer. If host and device have a shared memory (emulation), the `map` clauses do not issue an actual data transfer.

Since the OpenMP API does not only support GPU-like architectures as target devices, it has been a design decision by the OpenMP Language Committee to separate offload directives and parallelism from each other. Through this decision programmers can use the best matching OpenMP directives to

create parallelism for a specific target architecture. Also, the OpenMP API supports a more descriptive approach via the `loop` construct instead of the `teams distribute parallel for` construct.

The `teams distribute` directive then partitions the loop iteration space across the available warps or wavefronts, while the `parallel for simd` constructs can parallelize the partitioned loop for the available GPU threads. Another approach is to map `parallel for` to a single GPU thread and use `simd` to create parallelism within the warp/wavefront. OpenMP explicitly allows for this flexibility in laying out the execution on the GPU, such that implementations can pick the best possible strategy.

Many compilers now have (partial) support for version 5.0 and version 5.1 of the OpenMP API. In this work, we use only OpenMP offloading as we benchmark GPU accelerators. For AMD GPUs, we rely on the AMD OpenMP compiler (AOMP).

3.3 SYCL

SYCL [19] is an open standard for heterogeneous programming. It is developed and maintained by the Khronos Group. Unlike other heterogeneous programming models, SYCL does not introduce any custom syntax extensions or pragmas. Instead, expresses heterogeneous data parallelism with pure C++. The latest SYCL version is SYCL 2020, which relies on C++17. Originally, SYCL was intended as a higher-level single-source model for OpenCL. This means that in contrast to OpenCL, host and device code reside in the same source file in SYCL, and are processed together by the SYCL compiler. Starting with SYCL 2020, a generalized backend architecture was introduced that allows for other backends apart from OpenCL. Backends used by current SYCL implementations include OpenCL, Level Zero, CUDA, HIP and others.

While a more task-oriented model is available as well, SYCL currently strongly focuses on data parallel kernels. The execution of these kernels is organized by a task graph that is maintained by the SYCL runtime. There are two memory management models in SYCL: the buffer-accessor model and the unified shared memory (USM) model.

In the buffer-accessor model, the SYCL runtime handles data transfers automatically according to data access specifications given by the programmer. These are also used by the SYCL runtime to automatically construct a task graph for the execution of kernels. In the pointer-based USM model, the programmer is responsible for correctly inserting dependencies between kernels and making sure that data is available on the device when necessary. While the buffer-accessor model may introduce overheads due to the evaluation of the access specifications and calculation of kernel dependencies, if the scheduler receives detailed information that can be used to optimize the task graph execution.

The execution model in SYCL is largely inherited from OpenCL. Parallel work items are grouped into work groups, and synchronization is only possible within a work group. Starting with SYCL 2020, work groups are additionally subdivided into subgroups that are typically mapped to SIMD units. On GPUs,

a SYCL work group usually corresponds to a thread block from HIP or a team in the OpenMP model. As such, the SYCL work-group size is a tuning parameter as in those other models. In SYCL, multiple methods exist to invoke kernels. In the simplest method, `parallel_for`, the work groups are not exposed and, on GPUs, a SYCL implementation automatically selects an appropriate work group size. In the more complex `nd_range` model, the user is responsible for choosing an appropriate work group size.

There are multiple implementations of SYCL. The most well-known implementations include ComputeCpp [20], DPC++ [21], hipSYCL [22] and triSYCL [23]. In this work, we will be using hipSYCL as it has mature support both for the GPUs investigated in this work. hipSYCL consists of a multi-backend runtime with support for CPUs and GPUs from AMD, NVIDIA and Intel, the SYCL kernel and runtime header library, as well as a compiler component with a unified compiler driver called *syclcc*. This compiler component is designed to integrate with existing compiler toolchains. For example, when compiling for NVIDIA and AMD GPUs, hipSYCL acts as an additional layer on top of CUDA and HIP. During compilation, hipSYCL loads an additional clang plugin that extends clang’s native HIP and CUDA support with support for SYCL-specific constructs, such as automatic kernel detection and outlining. This design not only allows a user to mix-and-match CUDA or HIP kernel code with SYCL code even within one kernel, it also allows using vendor-supported toolchains with hipSYCL since e.g. AMD’s official ROCm HIP compiler uses the same clang HIP toolchain. Consequently, hipSYCL can be deployed on top of the AMD HIP compiler.

3.4 OpenACC

OpenACC is a directive programming model for the GPUs that has evolved significantly since its beginning. Initially, there were two options for OpenACC support on LUMI. First, the HPE/Cray compiler supports only Fortran and OpenACC version 2.7, with potential for up to v3.1 until end of 2022. Second, the GNU compiler [24], which is not a contractual agreement. Thus, our guidance is not recommending OpenACC without also mentioning these caveats.

For illustration, the following OpenACC directive uses a few clauses. The `gang` clause corresponds to the thread blocks, while the `worker` clause is the warp or wavefront, and `vector` is the threads:

```
#pragma acc parallel loop \
    copyin(A[:N]) copyout(B[:N]) \
    vector_length() gang worker num_workers()
...

```

As GCC with offload to AMD MI100 GPUs is not focus on performance this moment, but more to functionality, we do not report OpenACC results. We mention though that GCC v10.3, v11.1, and later have fixed an issue that GPU memory was cleaned too often and as a result the performance on NVIDIA GPUs is improved by almost 30% for all BabelStream kernels except the `dot`

kernel for which the performance remained similar. Moreover, in the future we plan to explore a research project called *clacc* [25,26] that provides OpenACC support for Clang and LLVM. This will allow for simplified porting of OpenACC codes to the OpenMP API (amongst other benefits).

3.5 Alpaka

The Abstraction Library for Parallel Kernel Acceleration (alpaka) [27] is implemented as a header-only C++14 abstraction library for accelerator development and portability. Originally developed to support large-scale scientific applications like PIconGPU [28], alpaka enables an accelerator-agnostic implementation of hierarchical redundant parallelism, that is, the API allows a user to specify data and task parallelism at multiple levels of compute and memory for a particular platform. Currently, alpaka provides support for backends for OpenMP, (C++) threads, Intel Threading Building Blocks, CUDA, HIP, and SYCL for FPGA along with new backends for directives in development.

Alpaka code can be used to express hierarchical parallelism for both CPU-style and GPU devices. In addition to grids, blocks, and threads, alpaka also provides an element construct that represents an n -dimensional set of inputs that is amenable to vectorization by the compiler. This extra level of parallelism is key to achieve good performance when attempting to map GPU-style kernels to a CPU architectures that offer SIMD instructions as part of their instruction set architecture.

In addition to the optimized kernels via alpaka, users can also use the C++ User interface for the Platform independent Library Alpaka (cupla) [29] to port CUDA code to use the alpaka library. Cupla codes have a very similar syntax to regular CUDA kernels and can include calls to the CUDA API for data allocation and movement. While cupla introduces some host-side API call overhead compared to pure alpaka, it provides a suitable path to map existing codes to alpaka's supported backends.

3.6 Kokkos

The Kokkos [30] C++ Performance Portability Ecosystem is a framework for writing modern C++ applications with portability across a variety of hardware. It is part of the Exascale Computing Project (ECP) and is used by many HPC users and packages. It supports several backends, such as CUDA, HIP, SYCL, and OpenMP offloading to target various accelerators, including NVIDIA and AMD GPUs.

The Kokkos abstraction layer maps C++ source code to the specific instructions required for the backend during build time. When compiling the source code, the binary will be built for the declared backends:

- Serial backend, for serial code on a host device.
- Host-parallel backend, which executes in parallel on the host device (OpenMP API, etc.).
- Device-parallel backend, which offloads on a device, such as a GPU.

4 Choosing a Programming Model

Figures 1 and 2 present the porting diagrams of potential codes targeting the LUMI system. Initially, developers make a decision based on whether the code is already able to use a GPU or not. If not (see Fig. 1), there is an option for the developer to try various programming models such as SYCL, Alpaka, or Kokkos if the application’s programming language is supported. Expert developers could port their code directly to HIP, identifying the kernels and preparing them similarly to CUDA. If the code does not have OpenMP directives, a tool such as Cray Reveal could be used to port the code to the OpenMP API. This procedure can be more productive for Fortran applications, and it could be expanded to the rest main programming languages later. Then the developer can manually port the OpenMP CPU code to user OpenMP `target` directives.

Then, a standard software tuning cycle can kick in. If the performance is not as expected and desired, then developers profile and tune the OpenMP directives, especially avoiding unnecessary data transfers. This cycle repeats until the problem is solved and the code works as expected. Otherwise, some OpenMP offload regions can be ported to HIP to expose more control over kernel execution. It should be mentioned that at the time of writing, the OpenMP implementation of AMD is composable with the HIP API, but requires to keep OpenMP code and HIP code in separate compilation units. If the code is in C/C++, then profile, identify the kernels, and port to HIP. If the code is CUDA Fortran, then it is required to use a Fortran interface for GPU kernels, called `hipfort` [31], to port the code to HIP. The developers could also use OpenACC instead of the OpenMP API once the compilers are available, but it depends on whether the applications are already using OpenMP directives or not, and what preference for the programming model is.

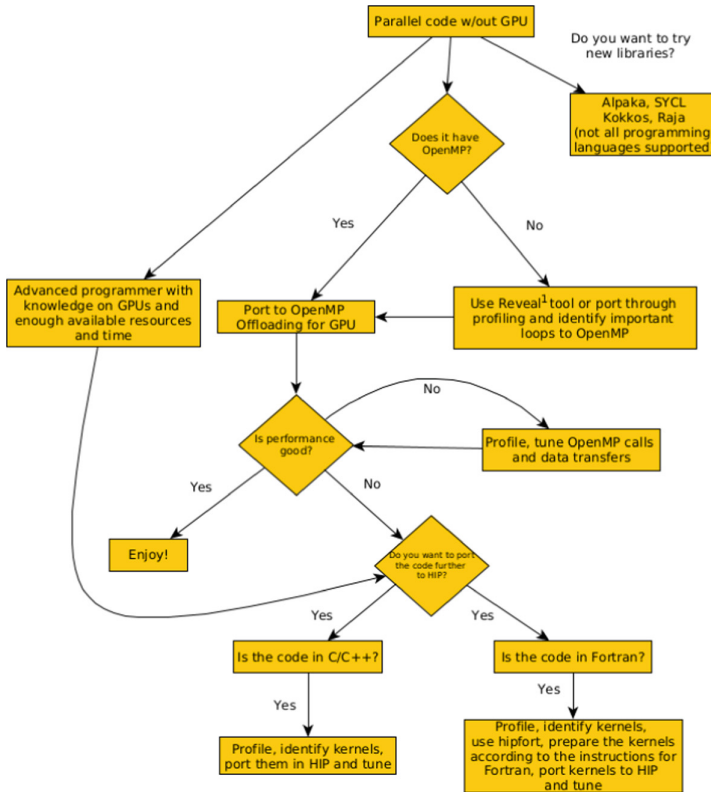
If the application is already ported to a GPU (see Fig. 2) there is a possibility to use the programming models such as SYCL, Alpaka, etc. If the initial application is developed in OpenACC, there are three options that are divided into sub-categories. First of all the Cray compilers are supporting only Fortran codes. LUMI is an HPE supercomputer, which means there is contractual engagement on the availability of some programming models. On the other side, the GCC efforts were mentioned in the OpenACC section before. Finally, the research projects such as Clacc, and Flacc which provides OpenACC support for Flang, are not yet in the final state. If the performance with any of the previous solutions is not good, then port the OpenACC calls to OpenMP to investigate the performance and tune the code.

If the GPU code is written in CUDA, and if it is C/C++ code it could be ported with the HIPify tools, while if it is in CUDA Fortran, then the `hipfort` should be used as also described in Sect. 3.1. Finally, if the performance is not as expected, a similar software tuning cycle is used to resolve the issue.

5 Benchmarks and Applications

5.1 BabelStream

BabelStream [32,33] is a memory bound benchmark with many programming models implemented. There are five computational kernels that we are using, the add ($a[i] = b[i] + c[i]$), multiply ($a[i] = b * c[i]$), copy ($a[i] = b[i]$), triad ($a[i] = b[i] + d * c[i]$), and dot ($sum = sum + a[i] * b[i]$). The default problem size is 2^{25} FP64 operations and 100 iterations. We are evaluating BabelStream v3.4 (6fe81e1). We developed the Alpaka backend for BabelStream for which we present some results in this paper.



¹ Reveal will work good with Fortran codes and less with C, especially C++

Fig. 1. Diagram for porting CPU applications to LUMI

5.2 MiniBUDE

We use also the mini-app called miniBUDE for the Bristol University Docking Engine (BUDE) [34], a kernel of a drug discovery application that is compute bound and provides performance results in single precision. BUDE is designed for in silico molecular docking. In the computationally intensive virtual screening, molecules of drug candidates, known as ligands, are bonded to target protein molecule. BUDE predicts the binding energy of the ligand with the target, however, there are many ways this bonding could happen, and a variety of positions and rotations of the ligand relative to protein, known as poses, are explored. And for each pose, a number of properties are evaluated. We are evaluating the version with commit 1af5b39.

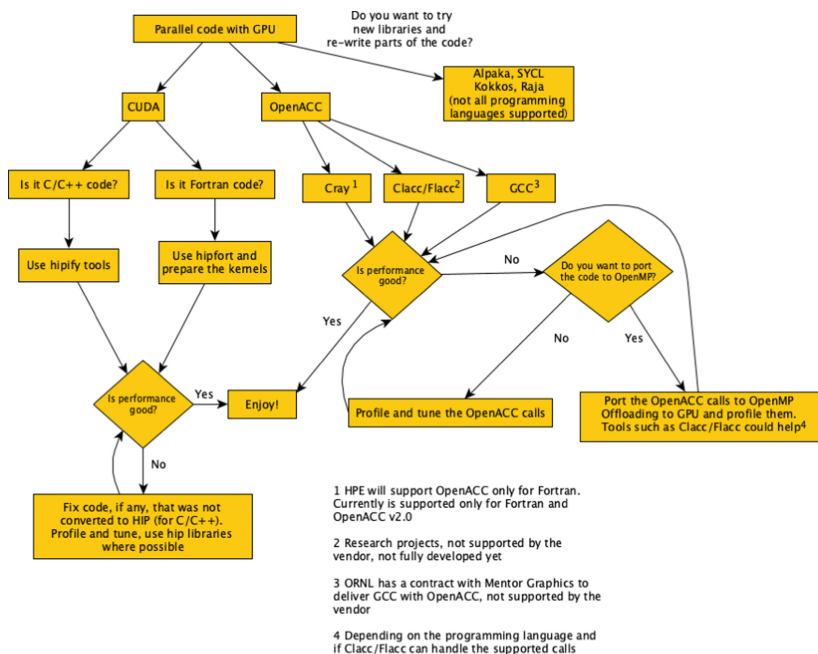


Fig. 2. Diagram for porting GPU applications to LUMI

Both BabelStream and miniBUDE support many programming models such as HIP, CUDA, OpenMP Offloading, SYCL, OpenACC, Kokkos.

6 Methodology

6.1 Compilation

For the compilation, we used the provided instructions from the benchmark and application. For the miniBUDE, we had some concerns about our installation

that we will discuss later as we could not achieve the expected performance but we had no issue with the BabelStream.

6.2 Execution and Tuning

We save the data from ten executions (in the same submission script, so using the same compute nodes). We then visualize the results in a box plot to determine variations and to ensure sure that our observations are correct. As our runs do not include MPI, we try to investigate if binding the processes helps in some cases or trying to have a process as close to the GPU as possible, but for our cases, we did not observe any significant improvement. We tune where possible by adjusting the number of the thread blocks to be a multiple of the number of compute units for the MI100 or streaming multiprocessors for V100/A100 GPUs.

7 Results

7.1 Configuration

We plan to utilize a single GPU, as we do not want to interfere with MPI performance evaluation in this work. CSC provides two supercomputers called Puhti and Mahti. The AMD Accelerator Cloud is a remote, heterogeneous system provided by AMD. Technical details about the GPUs specifications are presented in Table 2.

The **Puhti** [35] supercomputer at CSC, is constituted by 682 CPUs and 80 GPU nodes. Each GPU node has two Intel Xeon Gold 6230 processors with 20 cores each, and four NVIDIA V100 with 32GB HBM2 memory each. The interconnect is based on a dual-rail Mellanox HDR100 fabric.

The **Mahti** [36] supercomputer has 1404 CPUs and 24 GPU nodes. Each GPU node has two AMD EPYC™ 7H12 Processors (“Rome”) with 64 cores each, four NVIDIA A100 with 40 GB HBM2 memory for each one, and a total of 512 GB of memory.

Table 2. List of utilized GPU architectures and specifications

Vendor	Model	HBM Memory (GB)	MemoryBandwidth (GB/s)	Threads	Peak FP64 (TFLOPS)	Peak FP32 (TFLOPS)
NVIDIA	V100	32	900	5,120	7.8	15.7
NVIDIA	A100	40	1,555	6,912	9.7	19.5
AMD	MI100	32	1,200	7,680	11.5	23.1

The **AMD Accelerator Cloud** offers different GPU options. We used a node with two AMD EPYC 7742 Processors and four AMD Instinct MI100 accelerators.

In Table 3 we mention the compilers/software for each system that participated in our study and their versions.

7.2 BabelStream

In this subsection we present the results from the BabelStream and do comparisons between GPUs.

Versions and Generic Tuning

We mention some software versions and generic tuning that applies in most of the kernels below. If a kernel has a different tuning, it will be mentioned in the corresponding kernel. About HIP, we decrease the number of threads per block to 256 instead of 1024, and achieve on average a performance improvement of up to 28% than using the default number of threads. The AMD MI100 has 120 compute units, thus when the blocks of threads are a multiple of 120, are usually more efficient for this GPU because we hide the latency cost. It is known that the AOMP is under heavy development to achieve better performance. We work with one of the latest AOMP versions instead of building the LLVM from the provided AMD GitHub repository through ROCm, as the AOMP is closer to production. Moreover, according to our tests, all the kernels perform around 5% better between AOMP 13.x and AOMP 12.x except the *dot* kernel that it is improved with a factor of 2.7. One of the reasons is also that AOMP 13.x creates automatically two times more block of threads compared to version 12.x, including the AOMP performance improvements and the utilization of LLVM 13. A table which displays the range of the percentage of speedup of MI100 over V100 or the slowness of MI100 over A100 will be presented for each kernel. For all the experiments, hipSYCL uses HIP as backend for AMD GPUs and CUDA for NVIDIA GPUs. Finally, we developed the Alpaka backend for BabelStream [37].

Table 3. List of compilers

Compiler/Software	Version	System
AOMP	13.0-4-4	AMD accelerator cloud
LLVM	13.0.0	AMD accelerator cloud
ROCM/HIP	4.2	AMD accelerator cloud
NVIDIA HPC SDK	21.7	Puhti, Mahti
GCC	devel/omp/gcc-10 (6b88ea4)	Puhti
hipSYCL	0.9.1 (c759aac1d)	Puhti, Mahti, AMD accelerator cloud
Kokkos	3.4.1	Puhti, Mahti, AMD accelerator cloud
Alpaka (cupla)	commit 287deace	Puhti, Mahti, AMD accelerator cloud

Copy Kernel

Figure 3 demonstrates the results from the Copy kernel of the BabelStream across many programming models. On the x -axis are the names of the programming models and on the y -axis the bandwidth in GB/s is depicted. However, as we plot the boxplots, we split the y -axis in order to be able to visualize the plots more clearly. On each y -axis range, there are results only from a specific GPU whose name is mentioned on the right y -axis. In Table 4, we present in the three first rows the peak performance (in %) based on the best programming model and the last two rows demonstrate the percentage range of the differences, either slower for MI100 vs A100 or faster for MI100 vs V100. We observe that for all the GPUs, the HIP/CUDA programming models achieve the highest performance. Although the OpenMP performance seems to be close to the CUDA's one, thanks to efficient NVHPC compiler, the AMD OpenMP based on LLVM is not performing similar to HIP for this pattern. The default Kokkos implementation in BabelStream does not provide tuning options, however, we observe that for MI100 its results are close to not tuned HIP. Finally, hipSYCL has a variation on A100 which is less than 2%, and Alpaka achieves a performance similar to HIP for the MI100.

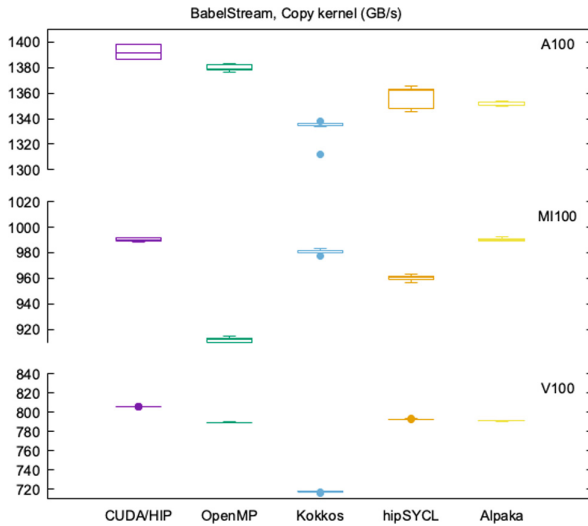


Fig. 3. Results of BabelStream for copy kernel across the programming models on AMD MI100, and NVIDIA V100/A100

Table 4. Copy kernel results, percentage peak and comparison (%)

	HIP/CUDA	OpenMP	Kokkos	hipSYCL	Alpaka
A100	100	98.5	96.3	97.53	97.3
MI100	100	92.6	99	97.8	99.99
V100	100	97.95	89.01	98.4	98.2
MI100 slower than A100	28.5–29.25	33–34	25.29–26.79	28.74–29.79	26.14–26.78
MI100 faster than V100	22.5–23	14.9–15.9	36.2–37.1	20.67–21.42	25.75–25.93

Multiply Kernel

We plot the results for the Multiply kernel in Fig. 4 and present the peak performance and the comparison in the Table 5. For MI100, most of the programming models achieve 96.63% and above except OpenMP, while for A100 all the programming models perform close to the peak, however, for V100, the not tuned Kokkos underperforms. For MI100, most programming models perform 21.6–37.8% faster than V100, except OpenMP, and similarly, MI100 is 25–31% slower than A100. For some cases there is variation up to 4.5% on A100 where for the moment we have not identified a specific reason as for all the cases we use a dedicate single node. However, it could be considered as execution variation.

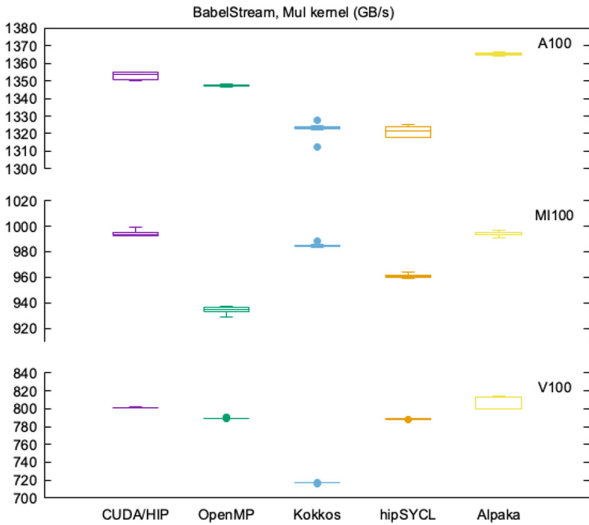


Fig. 4. Results of BabelStream for mul kernel across the programming models on AMD MI100 and NVIDIA V100/A100

Table 5. Multiply kernel results, percentage peak and comparison (%)

	HIP/CUDA	OpenMP	Kokkos	hipSYCL	Alpaka
A100	100	99.59	97.75	97.63	99.99
MI100	100	93.98	99.07	96.63	99.95
V100	100	98.54	89.52	98.39	99.90
MI100 slower than A100	26.16–26.78	30.4–30.98	24.92–25.79	26.99–27.57	26.96–27.44
MI100 faster than V100	23.8–24.74	17.8–18.6	37.0–37.8	21.6–22.22	21.74–24.52

Add Kernel

We plot the results for the Add kernel in Fig. 5 and present the peak performance and the comparison in the Table 6. The performance of Alpaka programming model is quite close to HIP/CUDA for all the devices with hipSYCL following, and the OpenMP is less efficient on the MI100 compared to the rest GPUs. Finally, Kokkos, seems to be between Alpaka and hipSYCL, regarding the performance.

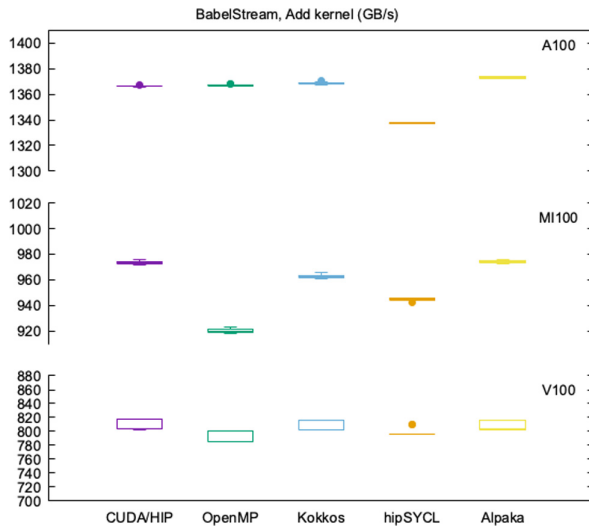


Fig. 5. Results of BabelStream for add kernel across programming models on AMD MI100 and NVIDIA V100/100

Table 6. Add kernel results, percentage peak and comparison (%)

	HIP/CUDA	OpenMP	Kokkos	hipSYCL	Alpaka
A100	100	99.80	97.6	97.70	99.99
MI100	100	94.52	98.92	97.06	99.99
V100	100	97.91	99.90	98.93	99.93
MI100 slower than A100	28.55–28.86	32.46–32.86	29.44–29.78	29.27–29.54	28.91–29.19
MI100 faster than V100	18.94–21.55	14.74–17.58	17.76–20.08	16.55–18.88	19.20–21.60

Triad Kernel

We plot the results for the Triad kernel in Fig. 6 and present the peak performance and the comparison in the Table 7. For this kernel Alpaka performs equally to HIP/CUDA, with following Kokkos, and then hipSYCL and OpenMP.

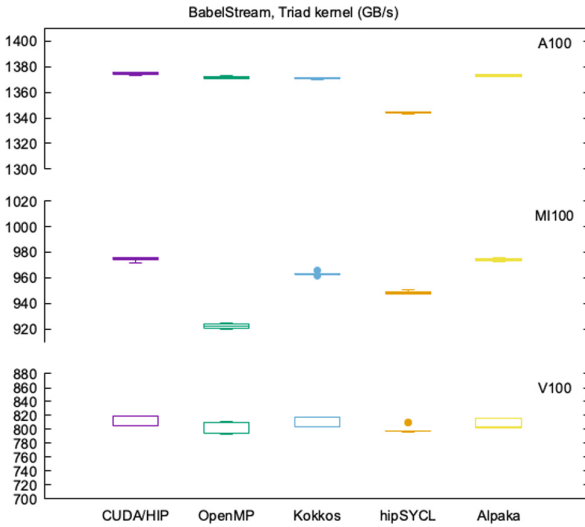


Fig. 6. Results of BabelStream for triad kernel across programming models on AMD MI100 and NVIDIA V100/A100

Dot Kernel

We plot the results for the Dot kernel in Fig. 7 and present the peak performance and the comparison in the Table 8. In the first segment of V100, we have also a

Table 7. Triad kernel results, percentage peak and comparison (%)

	HIP/CUDA	OpenMP	Kokkos	hipSYCL	Alpaka
A100	100	99.78	99.72	97.77	99.89
MI100	100	94.62	98.83	97.30	99.94
V100	100	98.82	99.86	98.80	99.69
MI100 slower than A100	28.93–29.31	32.6–32.81	29.52–29.87	29.21–29.57	28.91–29.19
MI100 faster than V100	18.63–21.25	13.47–16.40	17.79–20.13	17.00–19.25	19.02–21.61

plot of MI100 for hipSYCL as they were too close these values. MI100 GPU is around 2.69–14.62% faster than NVIDIA V100 except for OpenMP, and 28.00–69.69% slower than A100. The OpenMP on MI100 does not perform efficiently for the reduction pattern, and Kokkos is not optimized but it is quite close to non-optimized HIP version. For HIP/CUDA; We define 216 blocks of threads for the A100, as it has 108 streaming multiprocessors, and improved the *dot* kernel performance by 8–10%. For hipSYCL; we utilize 960 blocks of threads and 256 threads per block to achieve around 8% faster than the default values. For Alpaka; we are able to tune also the *dot* kernel with 720 blocks of threads for MI100 and improve its performance by 28% comparing to the default settings.

Table 8. Dot kernel results, percentage peak and comparison (%)

	HIP/CUDA	OpenMP	Kokkos	hipSYCL	Alpaka
A100	100	96.77	84.43	94.63	98.52
MI100	100	42.16	74.40	89.53	99.99
V100	100	96.5	72.87	96.95	99.38
MI100 slower than A100	29.09–29.84	68.7–69.69	37.19–38.43	32.67–33.65	28.00–28.62
MI100 faster than V100	10.80–11.86	-51.06 - (-51.513)	12.60–14.62	2.69–3.38	11.80–12.59

Summary

We can observe that based on the hardware performance AMD MI100 performs faster than NVIDIA V100 and slower than NVIDIA A100. The peak bandwidth percentage for the OpenMP programming model is 42.16%–94.68% for MI100 while it is at least 96% for the NVIDIA GPUs which demonstrates that AOMP needs further development. For Kokkos, the range is 74–99% for MI100, 72.87–99% for the NVIDIA GPUs, where the non-optimized version has lower

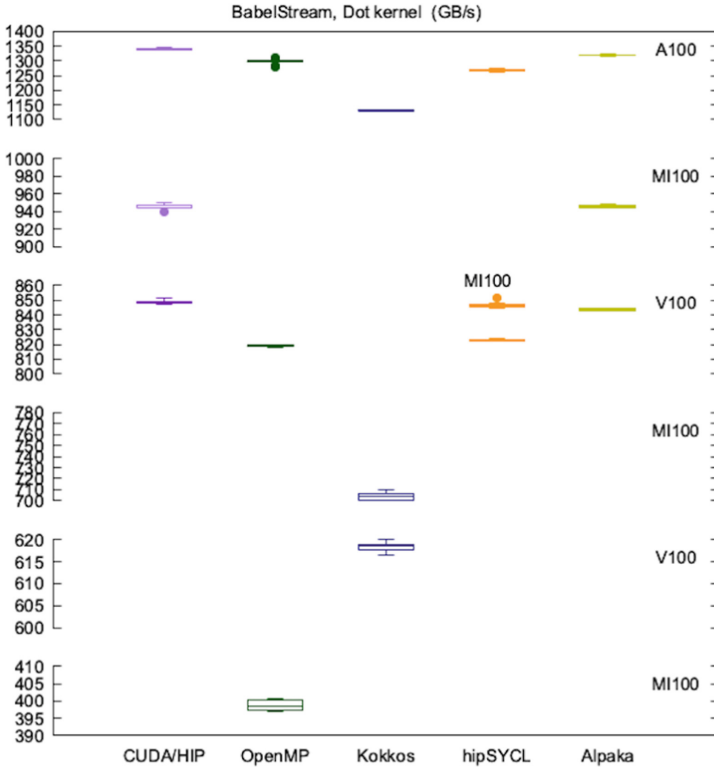


Fig. 7. Results of BabelStream with Kokkos HIP backend on AMD MI100 and NVIDIA V100/A100

performance mainly on MI100 and V100. hipSYCL achieves at least 96% of the HIP/CUDA performance except for MI100 and *dot* kernel that achieves 89.53%. Finally, Alpaka achieves at least 97.2% for all the cases and demonstrates its performance. The OpenMP compiler performs better for NVIDIA GPUs regarding *dot* kernel, however, the version that we used for AMD GPUs, is not the final product yet. The Kokkos results regarding the *dot* kernel are not optimized, we did not modify the execution policy and, we tried to use the default code and change only specific values, thus the low percentage. Also this demonstrates that some tuning are not so straight forward for Kokkos. Overall, the Alpaka performance is quite close to HIP, followed by hipSYCL and Kokkos, while OpenMP can perform slower depending on the kernel. We have to mention that all the remaining programming models except the OpenMP utilize HIP or CUDA as backend. The OpenMP Offloading for AMD GPUs has a potential to improve in the future as it is under development.

Finally, we should mention that we can observe that various programming models could have similar performance on the same GPU with some variation except OpenMP for some cases and Kokkos because is not optimized for some specific

cases. Overall, the tuning is not difficult if the developer is aware of the architecture and the programming model. Also, the utilization of each programming model depends on the experience of the developer, and the programming language as it was presented in the porting workflow.

7.3 MiniBUDE

Large problem sizes for miniBUDE are required to be able to saturate the GPUs. For every experiment, we execute 8 iterations with 983040 poses. We calculate this number by tuning for AMD MI100, however, this value achieves peak performance on the NVIDIA GPUs also with minimal variance of 1–2% and we decided to use the same workload for all the devices. The miniBUDE provides in the output the single precision GFLOP/s, and we observe in the Fig. 8 that the AMD MI100 GPU achieves a performance close to A100 by 2% and around 26% over V100. As the benchmark does not use tensor cores or other features, the peak performance is based on the FP32 capabilities of the GPUs. Thus, AMD MI100 is on average 1.25 times faster than NVIDIA V100, and 0.018 times slower than NVIDIA A100 for single precision using miniBUDE. For the moment, the other programming models do not perform very well, and we are still investigating the reasons. The code varies a bit between the programming models and the performance is significantly worse, thus we can not identify yet why both hipSYCL and Kokkos perform lower than HIP while using HIP for backend. Moreover, the Alpaka version for miniBUDE is under preparation. Regarding single precision, we tested also the mixbench [38,39] benchmark and the MI100 was 1.16 times faster than A100, achieving both close to their peak performance.

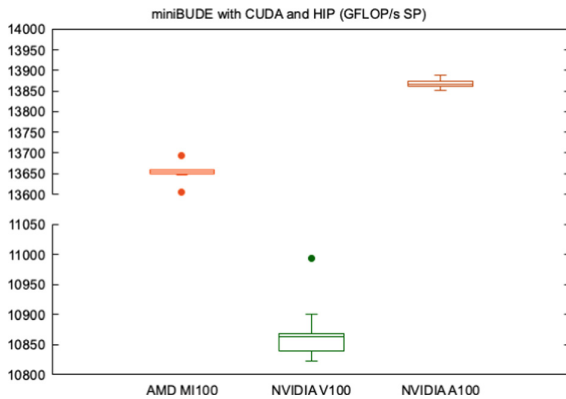


Fig. 8. Results of miniBUDE on various GPUs for HIP and CUDA

8 Conclusion and Future Work

In this paper, we present a methodology for porting applications to LUMI supercomputer, an AMD GPU-based system. As we expect many users to utilize LUMI, we are getting ready for a variety of porting scenarios. We benchmark various programming models to understand how they perform, how efficient they are, and which ones to propose to our future users. Thus, we do a performance comparison between AMD MI100, NVIDIA V100, and A100. We utilize a benchmark and a mini-app, which are memory and compute-bound respectively. We illustrate how various programming models perform on these GPUs and what techniques can improve the performance for specific cases. We discuss the lack of performance on some aspects of OpenMP, how to tune some programming models based on the targeted hardware and we verify the results. Moreover, the single precision mini-app demonstrates how similar performance to NVIDIA A100 has the AMD MI100 when not utilizing tensor cores. Overall, HIP/CUDA perform quite good and most of the programming models are quite close, depending on the kernel pattern. Depending on your experience, the programming language, and the kernel, you could leverage many of the programming models and always compare with the peak performance. Finally, programming models such as Alpaka and hipSYCL could be utilized as they support many backends, are portable and for many kernels they provide similar performance to HIP. All the scripts and the results are provided in [40] for reproducibility purposes.

For future work, we plan to identify what the issue with some programming models and miniBUDE is. We want to analyze the OpenACC performance from the GCC and Cray Fortran compiler, amongst tuning further the programming models, and to test the new functionalities from the ROCm platform such as Heterogeneous Memory Management (HMM). We envision evaluating multi-GPU benchmarks and their scalability across multiple nodes. By using LUMI we will be able to use the MI250X GPU and compare it with the current GPU generation. Finally, we are interested in benchmarking I/O from the GPUs memory as we have already hipified Elbencho [41] benchmark and a few applications could have significant I/O bottlenecks but we plan to benchmark them when LUMI is available as its architecture is much different to the available systems.

Acknowledgement. We want to thank CSC - IT Center for Science Ltd. for the access to Puhti and Mahti supercomputers. Tomas Tobias from Siemens for the discussions about GCC and LLVM. Finally, thank to Simon McIntosh-Smith and Wei-Chen Lin from University of Bristol for providing the necessary files to create new input problem sizes for miniBUDE.

This work was partly funded by the Center for Advanced Systems Understanding (CASUS) that is financed by Germany's Federal Ministry of Education and Research (BMBF) and by the Saxon Ministry for Science, Culture and Tourism (SMWK) with tax funds on the basis of the budget approved by the Saxon State Parliament.

Copyright 2021 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, EPYC, and Instinct and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

1. CSC LUMI supercomputer. <https://www.lumi-supercomputer.eu/lumis-full-system-architecture-revealed/>
2. Frontier web page. <https://www.olcf.ornl.gov/frontier/>
3. NVIDIA. CUDA. <https://developer.nvidia.com/about-cuda>
4. Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. In: Computing in Science & Engineering, vol. 12, no. 3, pp. 66–73, May–June 2010. <https://doi.org/10.1109/MCSE.2010.69>
5. OpenMP Architecture Review Board. OpenMP Application Programming Interface, version 4.0. <https://openmp.org/40pdf>
6. OpenACC Specification 3.0. <https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf>
7. Davis, J.H., Daley, C., Pophale, S., Huber, T., Chandrasekaran, S., Wright, N.J.: Performance assessment of OpenMP compilers targeting NVIDIA V100 GPUs. In: Bhalachandra, S., Wienke, S., Chandrasekaran, S., Juckeland, G. (eds.) WACCPD 2020. LNCS, vol. 12655, pp. 25–44. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-74224-9_2
8. Poenaru, A., Lin, W.-C., McIntosh-Smith, S.: A performance analysis of modern parallel programming models using a compute-bound application. In: 36th International Conference, ISC High Performance 2021, Frankfurt, Germany (2021)
9. Khalilov, M., Timoveev, A.: Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU. In: Journal of Physics: Conference Series, vol. 1740 (2021)
10. Deakin, T., McIntosh-Smith, S.: Evaluating the performance of HPC-style SYCL applications. In: Proceedings of the International Workshop on OpenCL (2020)
11. Deakin, T., et al.: Performance portability across diverse computer architectures. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 1–13 (2019). <https://doi.org/10.1109/P3HPC49587.2019.00006>
12. AMD. ROCm Platform. <https://github.com/RadeonOpenCompute/ROCm>
13. AMD. ROCm Documentation. <https://rocmdocs.amd.com/en/latest/>
14. AMD. HIP. <https://github.com/ROCm-Developer-Tools/HIP>
15. AMD. HIPify Tools. <https://github.com/ROCm-Developer-Tools/HIPIFY>
16. AMD. HIP Porting Guide. <https://github.com/RadeonOpenCompute/ROCm-Documentation/blob/master/Programming-Guides/HIP-porting-guide.rst>
17. CSC. Porting GPU Codes to HIP. <https://github.com/csc-training/hip>
18. de Supinski, B.R., et al.: The ongoing evolution of OpenMP. In: Proceedings of the IEEE, vol. 106, no. 11, pp. 2004–2019, November 2018
19. Khronos Group. SYCL 2020 Specification. <https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf>
20. Codeplay Software. ComputeCpp. <https://www.codeplay.com/solutions/ecosystem/>
21. Intel Corporation. SYCL* Compiler and Runtimes. <https://github.com/intel/llvm>
22. Alpay, A., Heuveline, V.: SYCL beyond OpenCL: the architecture, current state and future direction of hipSYCL. In: Proceedings of the International Workshop on OpenCL (IWOCL 2020), Association for Computing Machinery, New York, Article vol. 8, no. 1 (2020). <https://github.com/illuhad/hipSYCL>
23. triSYCL. <https://github.com/trisycl/trisycl>

24. ORNL and Mentor Graphics. <https://www.olcf.ornl.gov/2020/09/03/oak-ridge-leadership-computing-facility-fosters-gcc-compiler-development-with-mentor-contract/>
25. Denny, J.E., Lee, S. and Vetter, J.S.: Clacc: translating OpenACC to OpenMP in clang. In: 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC. LLVM-HPC), Dallas, TX, USA (2018)
26. Clacc. <https://github.com/llvm-doe-org/llvm-project/tree/clacc/main>
27. Zenker, E., et al.: Alpaka-an abstraction library for parallel kernel acceleration. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 631–640, May 2016
28. Bussmann, M., et al.: Radiative signature of the relativistic kelvin-helmholtz instability. In: SC 2013: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–12 (2013)
29. René, W., Sergei, B., Simeon, E., Jeffrey, K., Jan, S.: Cupla - C++ User interface for the Platform Independent Library alpaka. <https://rodare.hzdr.de/record/1103>
30. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. *J Parall. Distrib. Comput.* **74**, 3202–3216 (2014). <https://doi.org/10.1016/j.jpdc.2014.07.003>
31. AMD. hipfort. <https://github.com/ROCmSoftwarePlatform/hipfort>
32. Deakin, T., Price, J., Martineau, M., McIntosh-Smith, S.: GPU-STREAM v2.0: benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In: Paper presented at PMA Workshop at ISC High Performance, Frankfurt, Germany (2016). https://doi.org/10.1007/978-3-319-46079-6_34
33. Tom, D., Simon, M.-S.: BabelStream. <https://github.com/UoB-HPC/BabelStream>
34. miniBUDE. <https://github.com/UoB-HPC/miniBUDE/>
35. CSC. Puhti Supercomputer. <https://docs.csc.fi/computing/systems-puhti/>
36. CSC. Mahti Supercomputer. <https://docs.csc.fi/computing/systems-mahti/>
37. CUPLA BabelStream Fork, v3.4-alpaka release <https://github.com/jyoung3131/BabelStream/releases/tag/v3.4-alpaka>
38. Konstantinidis, E., Cotronis, Y.: A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. *J. Parall. Distrib. Comput.* **107**, 37–56 (2017)
39. Mixbench. <https://github.com/ekondis/mixbench>
40. Reproduce the results of the paper Evaluating GPU Programming Models for the LUMI Supercomputer. <https://zenodo.org/record/6307447>
41. Elbencho. <https://github.com/breuner/elbencho>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Evaluating Methods of Transferring Large Datasets

Jakub Kopec^(✉) 

Interdisciplinary Centre for Mathematical and Computational Modelling,
University of Warsaw, ul. Tyniecka 15/17, 02-630 Warsaw, Poland
jakubkopec1018@gmail.com
<https://icm.edu.pl/en/>

Abstract. Our society critically depends on data, Big Data. The humanity generates and moves data volumes larger than ever before and their increase is continuously accelerating. The goal of this research is to evaluate tools used for the transfer of large volumes of data. Bulk data transfer is a complex endeavour that requires not only sufficient network infrastructure, but also appropriate software, computing power and storage resources. We report on the series of storage benchmarks conducted using recently developed elbencho tool. The tests were conducted with an objective to understand and avoid I/O bottlenecks during data transfer operation. Subsequently Ethernet and InfiniBand networks performance was compared using Ohio State University bandwidth benchmark (OSU BW) and iperf3 tool. For comparison we also tested traditional (very inefficient) Linux scp and rsync commands as well as tools designed specifically to transfer large datasets more efficiently: bbcp and MDTMFTP. Additionally the impact of using simultaneous multi-threading and Ethernet jumbo frames on transfer rate was evaluated.

Keywords: I/O · File systems · Data management · Data transfer · File transfer protocols · Network evaluation · I/O benchmarking · Elbencho

1 The Outline of the Problem

The amount of data created and processed by humanity in the last few decades has grown exponentially. “Data explosion” is the term that is commonly used to describe this phenomenon. The Internet has evolved from the military project funded in 1965 [1] to the “place” where people spend large proportion of their time. Enormous amounts of data are generated which are sent over computer networks and stored in digital form on some sort of storage.

Historically Particle Physics and Astronomy were the major generators of Big Data generated by the particle accelerators and observatory equipment spread across the globe and beyond (e.g. The Hubble Telescope). Radioastronomy, with Square Kilometre Array (SKA) project - a radio telescope with a square kilometre collecting area located in Australia and South Africa will soon generate

observational data in volumes never imagined before. A prototype project: Australian Square Kilometre Array Pathfinder (ASKAP) is designed to be 1% the size of the full Square Kilometre Array and is to produce 60 Tb per second of raw data when finished [2]. Currently the ASKAP acquires 7,5 TB of data per second and the overall storage requirements of astronomical sciences are estimated to 1 EB per year.

However, Genomics might steal the title of the biggest data generator with the predictions of using 40 EB of storage per year by 2025. Among the factors that are boosting the amount of the data created by Genomics are: - the constant increase in number of the new high-tech sequencers in production; - increasing number of new endeavours to catalogue all the DNA of plants, animals and part of microbes; - and, of course, human Genomics - sequencing the human genome for the scientific and medical purposes. The huge data generators are the endeavours of sequencing the genomes of cancers that are characterised by unimaginable levels of genetic variations and the personalized medicine that will focus on the genome of an individual person [3].

All of these projects generate unimaginable amount of data that need to be stored and in many cases transferred over the computer network for analysis as the necessary computing power may not be available in situ. As the data transfer may be realised using various hardware and software the author decided to evaluate the performance of the chosen data transfer solutions.

The common misconception is that moving enormous amounts of data requires only appropriate network with sufficient bandwidth. For efficient bulk data transfers over a high-bandwidth network one needs powerful servers, highly performant file system and storage. Servers that are able to provide data at the rate that allows to saturate the link efficiently. It is impossible to use 100 Gbps network efficiently if the server provides data at 10 Gbps. The crucial aspect is the read/write speed of the storage device that is used in data exchange - the sending side must be able to provide (read) the data at appropriate speed and the receiving side must be able to ingest (write) incoming data. In the first place storage medium is to provide necessary data rate, but the server must also possess sufficient computing power and memory to cope with the transmission task. This hardware aspect is frequently neglected by the developers of data transfer solution and they tend to focus on the network bandwidth and software solution [4,5]. As storage device performance is critical to the data transfer endeavor we chose to start the evaluation with conducting storage benchmark with recently created *elbencho* tool [6] and storage sweep script that allows to gain good understanding of the storage system I/O throughput and characteristics [7].

In the era of cloud and Internet of Things (IoT) the approach of moving compute to the data, in order to reduce access costs, becomes a common practice. The example of such action is using microcontrollers located in the devices that acquire data (sensors, cameras, detectors etc.) to preprocess data stream before it is passed to the main compute system. In larger systems, e.g. in cloud applications part of the computation may be performed outside the cloud (at

the edge) and only the aggregated or abnormal results may be uploaded to the cloud for further analysis. Such actions limit the amount of the data that needs to be transferred [8]. However, there are numerous scenarios where this approach cannot be efficiently employed. Advanced scientific instruments, such as radio telescopes, particle detectors etc., are able to generate tremendous amount of raw data so they are already coupled with complex systems that reduce the data flow. But even after such reduction these are still large datasets that need to be transferred as a whole to remain meaningful and transferring them is inevitable for the backup reason alone, not to mention further sharing or processing them. Due to that the author believes that the need to move large datasets cannot be completely eradicated by the approach of moving compute to the data, thus the evaluation of existing transfer methods and finding new solutions of this problem remain relevant.

2 Tests Workbench: Hardware Specification

2.1 Servers

It is essential to treat the bulk data transfer as a complex issue that needs a holistic solution - appropriate servers interconnected with high-bandwidth network and the software that is able to efficiently utilise underlying infrastructure [4, 5]. To satisfy the balanced hardware requirements we used two HPE ProLiant DL385 Gen10 Plus servers that were customised to provide sufficient storage speed and compute power. The servers' specification is listed below:

- 2x AMD EPYC 7302 16-core (3.0 GHz) processors
- 16 HPE 1 × 32 GB Dual Rank x8 DDR4-3200 CAS-22-22-22 RAM memory (512 GB RAM memory in total)
- 8x HPE 3.84 TB NVMe Gen4 Mainstream Performance Read Intensive SFF SC U.3 CD6 SSD drives
- 2x 240 GB SATA SSD drives with HPE E208i-a SR GEN10 12G SAS controller
- Ethernet 100 Gb 2-port QSFP28 MCX516A-CCHT Adapter
- Intel I350-T4 Ethernet 1 Gb 4-port BASE-T OCP3 Adapter for HPE
- Mellanox MT27700 Family [ConnectX-4] InfiniBand Adapter

2.2 Mellanox InfiniBand Range Extenders

In the first part of the tests (storage benchmarking) the servers were interconnected directly with their 100 Gb InfiniBand Adapters, but in the second part of the test (network benchmarking) Mellanox's MetroX MTX6240 InfiniBand extenders were used. The vendor claims that this system is able to provide 40 Gbps throughput over 40 km of dark fiber [9] allowing to benefit from IB features (hardware-implemented RDMA) between geographically distributed sites [10–12]. MetroX system implements point-to-point communication - the two MTX6240 bundles, that consist of MEX6240 IB switch and MEX6200 DWDM

transponder, are located at the ends of the link - one bundle at each end. The MEX6240 IB switch may be connected to local IB network or directly to IB adapter in the server. During the tests the bundles were interconnected locally within one rack with short fiber (since longer dark fiber was not available at the time of tests) and the MEX6240 switches were connected directly to the IB adapters of HPE ProLiant servers.

3 Storage Benchmark - Elbencho

3.1 The Motivation for Storage Benchmarking

The storage performance is one of four crucial determinants that influence the overall data transfer performance - the storage device must be able to supply as well as ingest the transferred data at appropriate rate in order to saturate the connection and efficiently use available bandwidth. Vendors advertise their product's performance with the number of input/output operations per second (IOPS) or throughput (the amount of data transferred in a given time). Often these numbers may be quite meaningless as these may be the theoretical maximal values or the results obtained in the tests that simulate only narrow use scenario or even worse - are artificially designed to obtain high results without any respect to the real-life use of the drives. The hardware performance is closely coupled with the type of the executed workload - the same drive will perform differently when it will act as database with a lot of random writes/reads of small portions of data spanned across the whole medium than when it will read serial data sequence (i.e. streaming long video clip) saved in one physical location.

In order to evaluate the storage device one should run the tests that reflect the actual workload that will be run on a given device. As a production workload is not always available or possible to run as a test, people create benchmarks whose task is to simulate various workloads that may be spotted in real-life environments. Additionally the benchmarks may provide unified way to compare given devices or computer systems - the perfect example is the LINPACK benchmark that is used to measure performance and compare the computing power of the supercomputer systems for the TOP500 list [13].

In 2020 Sven Breuner created elbencho - a distributed storage benchmark for file systems and block devices with support for GPUs. Inspired by fio [14], mdtest and ior benchmarks [15] he wanted to create new, modern, easy to use and unified tool that may be used for testing storage systems performance. Elbencho allows testing the performance of GPU storage access. It has become essential nowadays as the deep learning and AI applications operate on GPUs [6]. The storage sweep script created by Chin Fang is invaluable part of elbencho toolbox as it provides the user with one-button-push ability to discover performance characteristics of a storage service with respect to the file size. It estimates the storage throughput by writing multiple hyperscale datasets (overall size bigger than 1 TB or number of files larger than 1 million) of different characteristics - a lots of small files (LOSF), an average amount of medium-sized files or a few of big files - and presents the results in a single csv file or on a graph

created with gnuplot [7]. There are other important aspects of benchmarking storage - the most noticeable are the testbed's configuration and the file size histogram of the test dataset. The hardware specification and the benchmark execution parameters should be as close to the projected production environment as possible - if the target application is host-oriented (e.g. database) then the benchmarking tool should be run on a single client host, analogically if the target application is cluster-oriented then the benchmark should be executed concurrently on multiple client hosts. Not only the servers specification should be identical to the target ones, but the whole system should be the same (e.g. storage, interconnects). The dataset structure (file size histogram) is important as transferring or processing a lot of small files is significantly slower than operating on smaller number of larger files [7] - this issue will be described in detail in the next section. "Storage sweeps", similarly to other benchmarks, should be carried out, as mentioned before, in a configuration that is intended for the target application. Nevertheless, as data transfer methods will be evaluated instead of production application, the "sweeps" will be performed using default options that are sufficient for comparison purposes [16].

3.2 Lots of Small Files Problem (LOSF)

Lots of small files (LOSF) issue is a common concern in any processing and transfer of the data. Every file is associated with the metadata - the data that describes the actual data that is valuable for the end-user, e.g. the location in the directory tree hierarchy (in case of file-based storage), the physical location on the storage device, the creation date, the last modification date, the owner, the access permissions etc. In the case of small files the metadata size may be comparable to the size of the actual data - hence the contribution of the overhead caused by the need of processing this metadata becomes significant part of any operation on the data. When the file size increases the ratio of the metadata size to the actual data size decreases and the excessive overhead diminishes. Other issues brought by the LOSF is the fact that any operation on file requires additional operations such as accessing the file, opening or closing it after processing. When dealing with the LOSF these operations are repeated frequently between the actual data reads/writes (that are rather short as there is not much data to process), but in the case of larger files these additional actions are less frequent and are separated with the significantly longer valuable data streams. The next problem may become less important with wider use of SSD drives, but it is still worth mentioning - a traditional storage devices (HDD) perform incomparably better operating on longer sequences of data that are stored in one physical location (track or adjacent sectors) than working on a lot of small files scattered randomly over the physical storage medium.

Here are a few examples of scientific domains in which the LOSF problem emerge as the datasets are most conveniently stored as independent files [17]:

- climatology - Community Climate System Model - 450k files with an average size of 61 MB [18],

- astronomy - Sloan Digital Sky Survey - 20 million files with an average size < 1MB [19],
- genomics - sequencing the human genome - 30 million files with an average size of 190 KB [20].

There are various attempts to efficiently address the LOSF issue, e.g. in machine learning some training sets are packed into single file. In the domain of data transfer one of the solutions is to compress the files into single archive and decompress it after transmission. Nevertheless, such solution is rather imperfect as compression and decompression require CPU time.

3.3 The Reasons for Conducting “Storage Sweeps”

There are three main reasons why we conducted “storage sweeps” in advance of the test of the actual data transfer methods. First of all, the storage I/O performance may be the one of the bottlenecks in the pipeline of data transmission tasks - the “sweeps” allow verification if the throughput of the storage service is sufficient to saturate the available network bandwidth. The results will also be useful to select the best file system to conduct the further research. Secondly, these “sweeps” are the ideal way of estimating the performance of the network file systems (NFS) implemented in various network configurations. Lastly, the “sweeps” are the good start for the tests of data transfer protocols as they bring useful information on how the successive tests may be carried out, for example which datasets should be used etc.

3.4 Storage Benchmarks Results

Local File Systems. The first series of tests concerned the local file systems and the results are plotted in Fig. 1. They helped to decide that for the successive tests the XFS file system will be used, as it’s performance was better in comparison with EXT4 - even though the maximal throughput was similar in both cases, with XFS the maximal value was obtained for the broader range of the datasets. The throughput of 8 SSD coupled in linux software RAID0 (using mdadm tool) was almost 8 times larger than the throughput of single SSD drive. We infer that the overhead of creating software RAID in this case was negligible. We conducted some tests of the BeeGFS distributed file system that is widely implemented on HPC systems [21]. As one may expect the BeeGFS Converged System Setup, which essentially means that all the required services (storage server, metadata server and client) are hosted within one physical server [22], cannot compete with the performance of the local XFS file system. It is caused by the overhead that results from the additional tasks required from distributed file system such as metadata synchronization or data replication. Moreover as all the services are launched on the single machine they may compete for resources which may create further overhead. In the conducted tests beeGFS gave approximately one

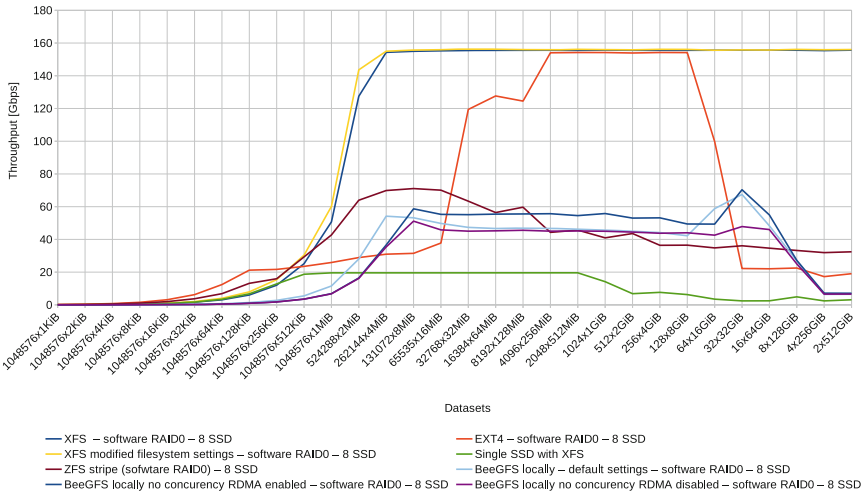


Fig. 1. Results of the “storage sweeps” conducted on local file systems

third of the throughput of XFS. Nevertheless, the distributed file systems are not designed to achieve maximal throughput per server, but to scale, i.e. to scale the performance proportionally to the large number of servers. In beeGFS documentation [23] ZFS file system was suggested as the appropriate base for beeGFS when using software RAID. The “sweeps” showed that ZFS-based software RAID’s performance is far below the performance of linux software RAID. In the case of locally accessed file system the RDMA feature did not influence the throughput which is not surprising. Hence the XFS will be the best choice of file system for the further tests of data transfer protocols/data movers and that the storage service in case of medium and large files will not be the bottleneck as 155 Gbps throughput will easily saturate the 100 Gbps network adapters installed in the servers.

Network File Systems. The first “network storage sweep” was run using 1 Gbps Ethernet adapter instead of 100 Gbps one. It showed that 1 Gbps link could be utilised in 100% without any effort and tuning using default settings. Further analysis of the results plotted in Fig. 2 show that traditional NFS cannot even obtain the throughput of 10 Gbps. The results obtained using NFS over 100 Gb Ethernet and NFS over IPoIB (IP packets encapsulated into InfiniBand packets and sent over IB network) are similar. For BeeGFS “sweeps” the elbencho was run on the server that was the BeeGFS client and the second server, that was connected to the first one with 100 Gbps Ethernet, acted as the BeeGFS storage and metadata server. The BeeGFS without using the RDMA feature performed slightly better than traditional NFS, but still it barely saturated 20% of the available bandwidth. The best results were obtained using the BeeGFS with RDMA feature enabled and NFS over RDMA (NFS using RoCE feature) [24].

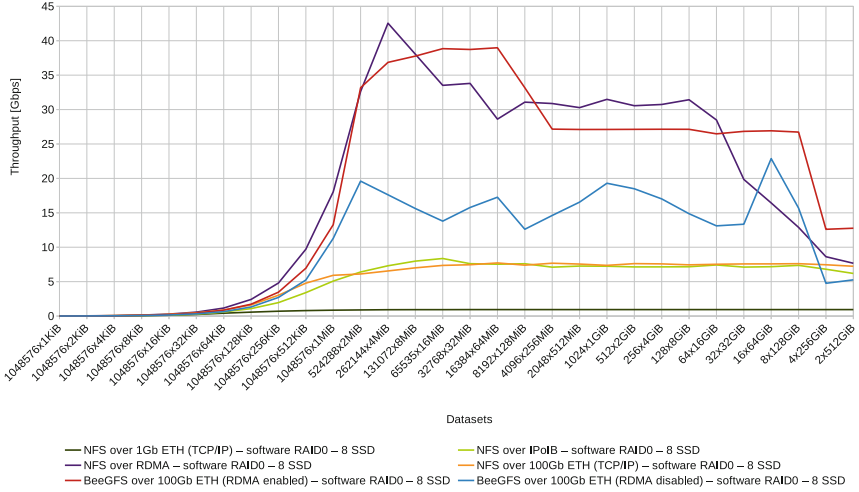


Fig. 2. Results of the “storage sweeps” conducted on network/distributed file systems

The RoCE-based NFS seems to perform slightly better, but still the results are comparable. However, any of the tested protocols/file systems did not allow to saturate so much as 50% of the link bandwidth. It is worth to notice that in all the tests the the throughput decreased for the datasets comprised of large files.

4 Network Benchmarks and Data Transfer Tests

Three types of the tests of data transmission methods were used:

- Ohio State University Bandwidth Test (OSU BW)

The OSU BW is a part of the OSU Micro-Benchmarks bundle developed at the Ohio State University in Network-Based Computing Laboratory. This benchmark may be used to test maximum data rate that is sustained in the network, it measures the bandwidth based on the transmission time of various-sized messages passed with non-blocking MPI functions [25].
- iperf3 network speedtest tool

Iperf3 is intended to measure the maximum achievable bandwidth on Internet Protocol-based networks and it’s development is mainly driven by the the U.S. Department of Energy Sciences Network (ESNet) and the Lawrence Berkeley National Laboratory [26].
- Sending the test elbencho-created datasets between the servers

The last series of test will consist of sending the test datasets, created using elbencho, between the servers installed in the testbed using various programs and comparing their performance.

4.1 Tests Methodology

During the tests one server acted as a host while the other one acted as a client. We decided to verify the impact of the AMD Simultaneous Multi-Threading (AMD SMT) feature on the data transfer rate - in Intel processors this feature is known as Hyper-threading and allows more efficient utilisation of CPU cycles. During the first test series the AMD SMT was disabled, and it was enabled in servers' BIOS for the second series.

We also wanted to verify how the utilisation of jumbo frames (the Ethernet frames with larger data payload) influence the transfer rates over the Ethernet. Since majority of the used applications is IP-based it was necessary to employ in such cases the Internet Protocol over InfiniBand (IPoIB) protocol that encapsulates IP packets into IB packets [27]. The use of the IPoIB is a major drawback, as it imposes additional overhead and almost completely eliminates the benefits brought by InfiniBand, but there is no other option of evaluating the performance of these applications with the MetroX IB extenders. Since the testbed operating system and IB adapter configuration did not allow the change of the IB frame size the influence of using jumbo frames with IPoIB protocol was not tested. As the out of the box server configuration usually is not able to provide full 100 Gbps throughput of the network interfaces the servers were appropriately tuned. The CPU governor was set to “performance” such as the power saving settings would not limit the CPU frequency. TCP maximal buffer size was extended to 2 GB, the maximum value possible in the Linux OS. Additionally the “fair queuing” (FQ) packet scheduler was used and the network interface interrupts were bound to the appropriate CPU socket using NIC vendor supplied script [28].

Ohio State University Bandwidth Test (OSU BW). The OSU BW benchmark is launched on two interconnected nodes simultaneously using MPI. We used MVAPICH [29] implementation of MPI to run the test. In each tested configuration the benchmark was launched three times and the average throughput of these runs is treated as an end result.

Iperf3. The tests with iperf3 measure overall network bandwidth. We also check if “zero copy” sending data method has any significant effect on the throughput. We checked if assigning the iperf3 process to appropriate CPU socket, so the affinity between the NIC and user process would be guaranteed, would impact the results. The test consisted of launching 6 instances of iperf3 simultaneously. The bandwidth of all iperf3 instances was aggregated and used as a test result. The use of single iperf3 instance was rather poor as each run in the same conditions resulted in significantly different outcomes. It is probably a result of the fact that iperf3 is single-threaded process and in the case of using high speed networks the CPU core frequency may become the bottleneck. The solution of this problem is launching multiple iperf3 instances that may utilise more than one CPU core [30]. Running 6 instances of iperf3 was optimal since adding more instances did not increase the aggregated throughput. The average of three consecutive runs is reported. This average is compared to the nominal bandwidth of the link.

Data Transfer of Elbencho-Created Datasets. This test comprised of transferring three test datasets using scp, rsync (standard Linux' tools to transfer data), bbcp [31] and MDTMFTP [32] in various testing configurations. The datasets were created using elbencho. It allows to create datasets containing random data with user-specified hierarchy (number of directories and number of files within them) and size. We created three test datasets that will correspond to three data structures:

- A lots of small files (LOSF) ($1024 \times 1 \text{ MiB} = 1 \text{ GiB}$)
- An average number of medium files ($40 \times 256 \text{ MiB} = 10 \text{ GiB}$)
- A few large files ($10 \times 10 \text{ GiB} = 100 \text{ GiB}$)

In case of scp and rsync the default parameters were used. bbcp enables user to tune many transfer parameters hence we decided to check if increasing the number of parallel TCP streams to 4 or using fixed-size optimal TCP window instead of auto-tuned one would improve the throughput. The suggestion of using 4 parallel TCP streams and the formula for optimal TCP window size - $(\text{bandwidth in Gbps})/8 * (\text{round-trip time between the source and target})$ - were found in the bbcp documentation [31]. The MDTMFTP and OSU BW are the only solutions implemented in this study that support InfiniBand natively (without the use of IPoIB) and are able to utilise it's full potential to saturate the links effectively. Moreover the MDTMFTP is multicore software that is able to utilise multiple cores [32]. The tests with MDTM required using larger LOSF and medium datasets as their transfer took few seconds which prevented throughput measurement - the transmission was so short that the MDTMFTP process was killed by the OS before reporting any results. For MDTMFTP tests the LOSF dataset and medium dataset were increased to approximately 10 GiB ($10000 \times 1 \text{ MiB}$) and 30 GiB ($120 \times 256 \text{ MiB}$) respectively. During the tests we noticed that using jumbo frames did not impact the transfer rates significantly, but instead it caused the MDTMFTP to crash repeatedly which precluded the transfer completion. For this reason we abandoned conducting the tests in the last configuration (AMD SMT enabled, transfer using jumbo frames) as this phase of tests was too time-consuming.

4.2 Results

Ohio State University Bandwidth Test (OSU BW). The results of OSU BW benchmark are shown in the graphs in Fig. 3. The first obvious observation is the fact that the results for all combination of parameters are practically the same - as this benchmark is supposed to test the maximum data rate in the network it is not surprising that servers' configuration (AMD SMT enabled/disabled, Ethernet frame size) does not affect the obtained results. Nevertheless, these graphs are an ideal way of visualisation the benefits of using InfiniBand as a transport protocol. The maximum data rate in 100 Gbps Ethernet is approximately 55 Gbps which means that the remaining 45% of the bandwidth is used by the service of the Ethernet protocol. On the other side

one may notice that 4xQDR InfiniBand maximum data rate is approximately 30 Gbps which means that almost 94% of the available bandwidth may be used to transfer valuable data. And that is the exact reason of the InfiniBand’s superiority over the Ethernet - it is able to efficiently saturate the network with meaningful data instead of congesting the fabrics with surplus control data.

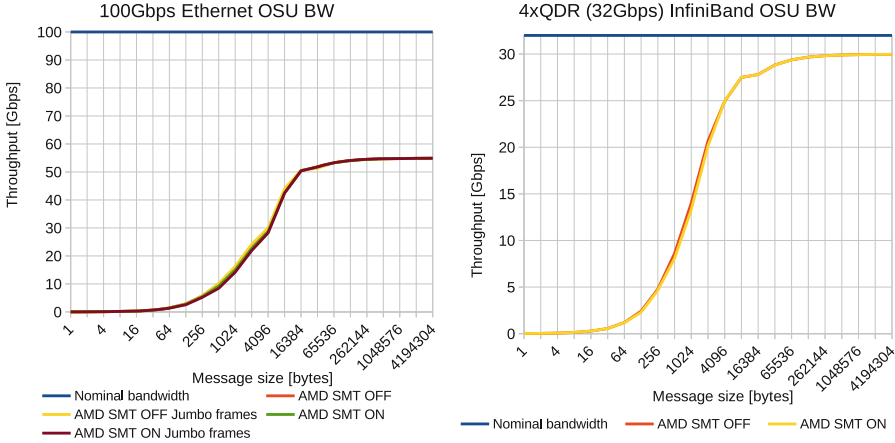


Fig. 3. Results of the OSU BW benchmarks with the additional plots of the links nominal bandwidths.

Table 1. The results of iperf3 benchmarks with AMD SMT disabled and with standardized Ethernet frames. All results are given in Gbps

	100 Gbps ethernet					4xQDR (32 Gbps) InfiniBand				
	Run 1	Run 2	Run 3	Average	% of 100 Gbps	Run 1	Run 2	Run 3	Average	% of 32 Gbps
No additional parameters	35,08	39,18	42,04	38,77	38,8%	14,18	14,18	14,19	14,18	44,3%
Zero copy	42,63	49,13	41,34	44,37	44,4%	11,84	11,84	11,85	11,84	37,0%
CPU affinity set	42,07	35,31	40,52	39,30	39,3%	9,52	9,52	11,90	10,31	32,2%

Iperf3. The results of iperf3 benchmarks are shown in Tables 1, 2, 3 and 4. By comparing the Tables 1 and 2 we notice that setting the CPU affinity does not seem to influence the throughput while using zero copy method of sending data boost up the throughput over a few Gbps. Turning on AMD SMT caused a drop in obtained throughputs. Such phenomenon is not seen in the results of the test with jumbo frames as in all cases the aggregated throughput achieve approximately the maximum network data rate of 55 Gbps.

In the iperf3 InfiniBand tests we do not find any significant correlations except for the fact that the achieved throughputs are a combinations of 2.38 Gbps

Table 2. The results of iperf3 benchmarks with AMD SMT enabled and with standardized ethernet frames. All results are given in Gbps.

	100 Gbps ethernet					4xQDR (32 Gbps) InfiniBand				
	Run 1	Run 2	Run 3	Average	% of 100 Gbps	Run 1	Run 2	Run 3	Average	% of 32 Gbps
No additional parameters	32,38	34,26	31,44	32,69	32,7%	14,22	9,52	11,90	11,88	37,1%
Zero copy	41,40	38,86	38,21	39,49	39,5%	14,23	14,10	14,23	14,19	44,3%
CPU affinity set	34,04	33,56	33,38	33,66	33,7%	9,52	11,90	14,25	11,89	37,2%

Table 3. The results of iperf3 benchmarks with AMD SMT disabled and with jumbo frames. All results are given in Gbps.

	100 Gbps ethernet				
	Run 1	Run 2	Run 3	Average	% of 100 Gbps
No additional parameters	54,50	54,87	54,81	54,73	54,7%
Zero copy	54,12	54,31	54,14	54,19	54,2%
CPU affinity set	55,46	55,34	55,50	55,43	55,4%

Table 4. The results of iperf3 benchmarks with AMD SMT enabled and with jumbo frames. All results are given in Gbps.

	100 Gbps ethernet				
	Run 1	Run 2	Run 3	Average	% of 100 Gbps
No additional parameters	54,54	54,86	54,96	54,79	54,8%
Zero copy	54,12	54,92	53,87	54,30	54,3%
CPU affinity set	55,42	55,25	55,20	55,29	55,3%

Table 5. The results of data transfer tests with AMD SMT disabled and with standardized Ethernet frames. All results are given in Gbps.

	100 Gbps ethernet				4xQDR (32 Gbps) InfiniBand			
	LOSF	Medium	Large	% of 100 Gbps	LOSF	Medium	Large	% of 32 Gbps
scp	0,95	1,11	1,56	1,6%	0,92	1,36	1,71	5,3%
rsync	1,12	1,55	1,42	1,5%	1,12	1,58	1,44	4,9%
bbcp	2,30	13,60	16,80	16,8%	1,92	7,91	8,80	27,5%
Bbcp - optimal window size	1,96	13,60	16,00	16,0%	1,51	7,88	8,80	27,5%
bbcp - 4 streams	2,30	12,80	16,00	16,0%	1,88	7,92	8,80	27,5%
MDTMFTP	28,23	27,87	28,41	28,4%	17,08	20,19	25,60	80,0%

and 1.19 Gbps (a half of 2.38) obtained by the individual iperf3 instances. It may suggest that IPoIB imposes some limit on encapsulated frames that causes the repetitiveness of per-thread results. There is no evident impact of setting the CPU core affinity between the NIC and user process as the obtained throughput was similar to the outcomes of the tests conducted with default iperf3 parameters. Possibly the obtained throughput was too small to benefit or bring loss from the CPU affinity settings.

Table 6. The results of data transfer tests with AMD SMT enabled and with standardized Ethernet frames. All results are given in Gbps.

	100 Gbps ethernet				4xQDR (32 Gbps) InfiniBand			
	LOSF	Medium	Large	% of 100 Gbps	LOSF	Medium	Large	% of 32 Gbps
scp	1,11	1,34	1,47	1,5%	0,89	1,44	1,40	4,5%
rsync	1,12	1,52	1,61	1,6%	1,12	1,74	1,58	5,4%
bbcp	2,19	12,80	18,40	18,4%	1,91	7,91	8,80	27,5%
bbcp - optimal window size	1,93	13,60	15,20	15,2%	1,48	7,89	8,80	27,5%
bbcp - 4 streams	2,24	13,60	16,00	16,0%	1,85	7,92	8,80	27,5%
MDTMFTP	27,40	29,95	30,65	30,6%	16,71	18,99	27,96	87,4%

Table 7. The results of data transfer tests with AMD SMT disabled and with jumbo frames. All results are given in Gbps.

	100 Gbps ethernet			
	LOSF	Medium	Large	% of 100 Gbps
scp	0,95	1,18	1,63	1,6%
rsync	1,12	1,40	1,69	1,7%
bbcp	2,21	12,80	16,00	16,0%
bbcp - optimal window size	2,10	12,80	16,00	16,0%
bbcp - 4 streams	0,22	12,80	16,00	16,0%
MDTMFTP	—	—	—	—

Data Transfer Tests. The results of the data transfer tests are listed in Tables 5, 6 and 7 and the first conspicuous fact is that how poorly the standard Linux data transfer tools (scp, rsync) utilise available bandwidth - in all configurations they were not able to provide as much as 2 Gbps throughput. While using the 100 Gb Ethernet they used approximately 1,6% of the bandwidth. In the case of the InfiniBand they used approximately 5% of the bandwidth, but that fact is meaningless as it does not result from the increase in the achieved throughput, but from the decrease of the available bandwidth. Regardless of the testbed configuration the results obtained by scp and rsync were similar and oscillated around 1,34 Gbps. This poor performance of these tools is probably caused by the fact that these tools use OpenSSH with built-in 1 MB buffer to encrypt the transferred data [33]. In order to remove that bottleneck one should look for tool that uses another encryption protocol or change the “data mover” to one that encrypt only control channel and sends the actual data unencrypted (which is acceptable in some applications) - for example bbcp [34]. The result obtained with bbcp shows that auto-tuned parameters are optimal as any attempt of manual tuning caused the slight decrease of the throughput or did not bring any positive effect. The bbcp results show perfectly the issue of processing the LOSF as the throughput obtained when transferring large files was approximately 8–9 times bigger than in the case of the LOSF. In all test conducted with bbcp we noticed that there seems to be a limit on the maximal throughput that may be obtained using this program - 16 Gbps on the Ethernet and 8,8 Gbps on the InfiniBand. We believe that this limitation may arise from the fact that bbcp is single-thread program and CPU frequency is the factor that limits

the throughput. However, in the case of InfiniBand this limit may be a result of the IPoIB encapsulation. The most interesting are the results obtained with MDTMFTP - its mechanism of dealing with the LOSF problem proved to be successful as the differences between the LOSF and large files throughput were not only significantly smaller, but on the Ethernet it seemed to disappear completely. While transferring the large files using IB the MDTMFTP was able to saturate approximately 80–90% of the available bandwidth that was impossible to achieve with any other tested software. The change of the size of the Ethernet frame did not result in any major change of the achievable throughput, but it only caused the instability of MDTMFTP software - the numerous errors prevented obtaining any reliable results of MDTMFTP performance with the use of the jumbo frames. Enabling AMD SMT feature revealed slight improvement of the throughput obtained with the MDTMFTP, no other changes were noticed.

4.3 Additional Comments on the Tests Results

We were not able to notice any significant impact of enabling AMD SMT feature (except for small decrease of throughput in the iperf3 tests and slight improvement in MDTMFTP transfer rates) that would allow drawing unambiguous conclusions on its influence on transfer rates. The usage of the jumbo frames undoubtedly improved the obtained throughput (what was observable in iperf3 tests), but none of the evaluated “data movers” can benefit from that increase as single-threaded applications were not able to utilise such bandwidth and MDTMFTP became unstable and the jumbo frames caused numerous errors and crashes.

5 Conclusions

The tests revealed striking inefficiency of the most popular Linux transfer tools on high-bandwidth networks. These tools were developed when the volume of transferred data and network bandwidths were incomparably smaller, thus they are not able to perform efficiently with the current volumes of transmitted data. Their design and underlying protocols are not able to saturate modern high-speed network links - approximately 98% of the bandwidth was wasted. This software may still perform well in the situations it was designed for. Scp is a useful tool to transfer few gigabytes over 100Mbps residential network, but it is by far not sufficient and outright wasteful to transfer hundreds of terabytes of scientific data across intercontinental 100Gbps link. The test has also shown how much bandwidth capacity may be spared by using the InfiniBand fabrics instead of the Ethernet. The InfiniBand is able to utilise efficiently more than 90% of the bandwidth while the Ethernet barely uses half of it after thorough tuning and effort. But the most importantly this research allows to understand how complex an issue the efficient transfer of digital data is and, that the network bandwidth is only one part of the mix and to transfer data efficiently one needs also appropriate storage, file system and computing resources, not to mention the suitable software.

5.1 Future Work

This research by no means did exhaust a topic of bulk data moving as it is very broad and complex problem that exists as the computer networks evolve and the number of links and their bandwidth increase rapidly. Moreover the storage technology is advancing rapidly as the new media are being developed. This study was focused on evaluation of common hardware and widely available software performance, however, there are several areas where the further research may be conducted.

New Storage Media. As the new non-volatile memory technologies emerge, such as spin-torque transfer RAM (STT-RAM), phase-change (PCM) and resistive (ReRAM) memory [36,37] or Intel's 3D XPoint [37,38] their performance could be assessed. For instance, the new Intel Optane SSD drives that employ 3D XPoint memory technology could be benchmarked using *elbencho* or its influence on the data transfer rate may be evaluated.

NVM-adapted File Systems. With the advent of fast non-volatile memories with the DRAM-like performance and byte-addressability current file systems are becoming the new performance bottleneck. Modern journaling file systems are designed to use whole data block as the basic unit of the journal what may cause significant overhead. As the actual price of the next generation NVM prohibits it to be used as standalone large-capacity memory device it may be beneficial to use it in hybrid DRAM/NVMM (non-volatile main memory) or NVDIMM solutions where NVMM may act as the external journal device for journaling files systems [37,39,40] or the journaling strategies may be altered to be more suitable for NVM devices [41]. On the other hand maybe the new principles of designing file systems are required as in [42,43] to fully utilize the cutting-edge NVM media. The evaluation of mentioned solutions may be the further extension of this research.

Alleviate OpenSSH Bottleneck Using HPN-SSH. The poor performance of SSH-backed tools caused by the limited size of OpenSSH buffer and the significant encryption overhead may be improved by using HPN-SSH - a research project that consists of a series of modifications to OpenSSH created at the Pittsburgh Supercomputing Center. Authors of the modifications reported that the throughput of SSH was at least doubled while using their tuning [44]. The data transfer tests of SSH-backed tools could be repeated with the use of HPN-SSH for the comparison purposes.

Creating Geographically Distributed Testbed. Developing further geographically distributed systems enabled with the MetroX InfiniBand or Vcinity range extenders [35] and comparing its performance with Ethernet would allow for validation of obtained results in productional environment.

Evaluating Other Software Solutions' Performance. During this research only the widely available software was evaluated. In the upcoming studies the performance of other software, for instance XRootD [45] may be evaluated. If possible the performance of proprietary solutions, such as Zettar's zx [46] or Obsydian Strategies' dsync+ [10] could be investigated.

Acknowledgments. The work reported here constituted research part of my Master of Science degree in Computational Engineering undertaken at the Interdisciplinary Centre for Mathematical and Computational Modelling,

University of Warsaw (ICM UW) under supervision of Dr Marek Michalewicz who recommended this topic and guided me throughout. I would like to express my gratitude to people who contributed to the realisation of this project: Chin Fang (Zettar Inc.) and Marcin Semeniuk (ICM UW) for sharing their knowledge with me and their valuable comments on my work; and to Bartosz Drogosiewicz (ICM UW) and Jarosław Skomial (ICM UW) for helping me build the testbed for this project. I also wish to thank Hewlett Packard Enterprise Polska Sp. z o.o. for providing two test HPE ProLiant DL385 Gen10 Plus servers used in this study.

References

1. Kleinrock, L.: An early history of the internet [History of Communications]. IEEE Commun. Mag. **48**(8), 26–36 (2010)
2. Newman, R., Tseng, J.: Memo 134 Cloud Computing and the Square Kilometre Array (2011)
3. Stephens, Z., et al.: Big data: astronomical or genomical? PLOS Biol. **13**(7), 1–11 (2015)
4. Fang, C.: Moving massive amounts of data across any distance efficiently, Talk on 2020 Rice Oil & Gas HPC Conference. <https://www.youtube.com/watch?v=8PCjMSKMyRw>. Accessed 17 Apr 2021
5. Zettar Inc. white paper: Understanding moving data at scale & speed (2020). <https://www.zettar.com/white-paper/>. Accessed 01 Mar 2021
6. Breuner, S.: elbencho github repository. <https://github.com/breuner/elbencho/>. Accessed 29 May 2021
7. Fang, C.: Storage Sweep github repository. https://github.com/breuner/elbencho/tree/master/contrib/storage_sweep. Accessed 29 May 2021
8. Near-Memory Computing. https://semiengineering.com/knowledge_centers/compute-architectures/near-memory-computing/. Accessed 02 Dec 2021
9. Mellanox TX6240 product brief. https://www.mellanox.com/related-docs/prod_long_haul_systems/MetroX_TX6240.pdf. Accessed 29 Mar 2021
10. Michalewicz M., et al.: InfiniCortex: concurrent supercomputing across the globe utilising trans-continental infiniband and galaxy of supercomputers. In: Supercomputing 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans (2014). <https://doi.org/10.13140/2.1.3267.7444>
11. Michalewicz, M.: InfiniCortex: present and future (2020). <https://doi.org/10.1145/2903150.2912887>
12. Niedziewski, K., et al.: Long distance geographically distributed InfiniBand based computing (2020). <https://doi.org/10.14529/jsfi200202>

13. About The Linpack Benchmark webpage. <https://www.top500.org/project/linpack/>. Accessed 29 May 2021
14. Flexible I/O Tester (fio) github repository. <https://github.com/axboe/fio>. Accessed 29 May 2021
15. IOR and mdtest github repository. <https://github.com/hpc/ior>. Accessed 29 May 2021
16. Fang, C., Cottrell, R., Kissel, E.: When to use rsync - DOE Technical Report. <https://slac.stanford.edu/pubs/slactns/tn06/slac-tn-21-001.pdf>. Accessed 29 May 2021
17. Carns P., Lang S., Ross R., Vilayannur M., Kunkel J., Ludwig T.: Small-file access in parallel file systems. In: 2009 IEEE International Symposium on Parallel & Distributed Processing, pp. 1–11. IEEE, Rome (2009). <https://doi.org/10.1109/IPDPS.2009.5161029>
18. Chervenak A., et al.: Monitoring the Earth System Grid with MDS4. In: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing, p. 69. IEEE, Amsterdam (2006). <https://doi.org/10.1109/E-SCIENCE.2006.261153>
19. Neilsen, E.H., Jr.: The Sloan digital sky survey data archive server. *Comput. Sci. Eng.* **10**(1), 13–17 (2008)
20. Bonfield, J.K., Staden, R.: ZTR: a new format for DNA sequence trace data. *Bioinformatics* **18**(1), 3–10 (2002)
21. Why Use BeeGFS webpage. <https://www.beegfs.io/c/home/why-use-beegfs/>. Accessed 29 May 2021
22. BeeGFS Documentation, Architecture overview. <https://doc.beegfs.io/latest/architecture/overview.html>. Accessed 29 May 2021
23. Tips and Recommendations for BeeGFS Storage Server Tuning webpage. <https://www.beegfs.io/wiki/StorageServerTuning>. Accessed 29 May 2021
24. HowTo Configure NFS over RDMA (RoCE) webpage. <https://community.mellanox.com/s/article/howto-configure-nfs-over-rdma-roce-x>. Accessed 07 Apr 2021
25. OSU Micro-Benchmarks webpage. <http://mvapich.cse.ohio-state.edu/benchmarks/>. Accessed 09 Jun 2021
26. iperf3 homepage. <https://iperf.fr/>. Accessed 09 Jun 2021
27. Kashyap, V.: RFC 4392 - IP over InfiniBand (IPoIB) Architecture. <https://www.rfc-editor.org/rfc/rfc4392.html>. Accessed 09 Jun 2021
28. ESnet FASTERdata Knowledge Base - 40G/100G Tuning webpage. <https://fasterdata.es.net/host-tuning/linux/100g-tuning/>. Accessed 09 Jun 2021
29. MVAPICH homepage. <http://mvapich.cse.ohio-state.edu/>. Accessed 09 Jun 2021
30. ESnet FASTERdata Knowledge Base - iperf3 at 40Gbps and above webpage. <https://fasterdata.es.net/performance-testing/network-troubleshooting-tools/iperf/multi-stream-iperf3/>. Accessed 09 Jun 2021
31. BBCP homepage. <https://www.slac.stanford.edu/~abh/bbcp/>. Accessed 16 Mar 2021
32. MDTM Homepage. <https://mdtm.fnal.gov/index.html>. Accessed 17 Mar 2021
33. ESnet FASTERdata Knowledge Base - scp and sftp webpage. <https://fasterdata.es.net/data-transfer-tools/scp-and-sftp/>. Accessed 09 Jun 2021
34. ESnet FASTERdata Knowledge Base - Data transfer tools background webpage. <https://fasterdata.es.net/data-transfer-tools/background/>. Accessed 09 Jun 2021
35. WAN Interoperability Overview - Vcinity Application Note. https://vcinity.io/sites/default/files/WAN_Interop_AN_RevA.pdf. Accessed 29 Mar 2021

36. Suzuki, K., Swanson, S.: The Non-Volatile Memory Technology Database (NVMDB). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego (2015)
37. Xu J., Swanson, S.: NOVA: a log-structured file system for hybrid volatile/Non-volatile main memories. In: 14th USENIX Conference on File and Storage Technologies (FAST 16), pp. 323–338. USENIX Association, Santa Clara (2016)
38. Intel and Micron Produce Breakthrough Memory Technology. <https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/>. Accessed 16 Feb 2022
39. Chen, C., Yang, J., Wei, Q., Wang, C., Xue, M.: Fine-grained metadata journaling on NVM. In: 32nd International Conference on Massive Storage Systems and Technology (MSST 2016), pp. 1–13 (2016). <https://doi.org/10.1109/MSST.2016.7897077>
40. Xu J., et al.: NOVA-Fortis: a fault-tolerant non-volatile main memory file system. In: Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017), pp. 478–496. Association for Computing Machinery, New York (2017). <https://doi.org/10.1145/3132747.3132761>
41. Chen, C., Wei, Q., Wong, W.-F., Wang, C.: NV-journaling: locality-aware journaling using byte-addressable non-volatile memory. *IEEE Trans. Comput.* **69**(2), 288–299 (2020). <https://doi.org/10.1109/TC.2019.2948004>
42. Kwon, Y., Fingler, H., Hunt, T., Peter, S., Witchel, E., Anderson, T.: Strata: a cross media file system. In: Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017), pp. 460–477. Association for Computing Machinery, New York (2017). <https://doi.org/10.1145/3132747.3132770>
43. Lu, Y., Shu, J., Chen, Y., Li, T.: Octopus: an RDMA-enabled distributed persistent memory file system. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17), pp. 773–785. USENIX Association, Santa Clara (2017)
44. Rapier, C., Bennett, B.: High speed bulk data transfer using the SSH protocol. In: Proceedings of the 15th ACM Mardi Gras conference (MG 2008), Association for Computing Machinery, New York (2008). <https://doi.org/10.1145/1341811.1341824>
45. XRootD Homepage. <https://xrootd.slac.stanford.edu/index.html>. Accessed 16 Mar 2021
46. A Software Engine for Moving Data at Scale & Speed - Zettar product brief. https://www.zettar.com/wp-content/uploads/2020/10/Zettar_product_brief.pdf. Accessed 01 Mar 2021





Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Service Function Chaining Design & Implementation Using Network Service Mesh in Kubernetes

Abdullah Bittar¹, Ziqiang Wang¹, Amir Aghasharif¹,
Changcheng Huang¹, Gauravdeep Shami², Marc Lyonnnais²,
and Rodney Wilson²

¹ Carleton University, Ottawa, ON K1S 5B6, Canada
{abdullahbittar, ziqiangwang, amiraghasharif, huang}@carleton.ca

² Ciena Corporation, Ottawa, ON K2K 0L1, Canada
{gshami, mlyonnai, rwilson}@ciena.com

Abstract. Service Function Chaining (SFC) in a cloud-native environment is becoming essential as more users move towards clouds today. Cloud-native environments utilize container-based microservices to provide software solutions. Integrating SFC with container-based microservices introduces new challenges. This paper exploited Network Service Mesh (NSM) framework features to create a service function chain on a multi-node Kubernetes cluster. We focus on the design and implementation of SFC in Kubernetes using NSM. Also, we deployed our custom-built containers in the Kubernetes cluster to create a service function chain. Hence, we demonstrate how an SFC is designed in a cloud-native environment rather than a traditional NFV/SDN approach. Furthermore, to evaluate the performance, we compare different frameworks that support SFC in Kubernetes, highlighting the advantage and disadvantages of each framework.

Keywords: Service Function Chain · SFC · Kubernetes · Network Service Mesh · NSM · Design · Implementation

1 Introduction

Next-generation networks mainly rely on the virtualization of network functions [1]. The network virtualization concept affects network operations, deployment, and expansion, especially by leveraging its benefits. Network Function Virtualization (NFV) provides some benefits such as scalability, flexibility, and cost. The idea behind NFV is to integrate different network equipment into industry-standard high-capacity servers, storage, and switches that can be located in various locations, such as data centers [2].

NFV introduced new opportunities to exploit the network and provide better services for its users. According to the International Data Group (IDG)

2020 survey, 92% of the organization’s IT environments are at least somewhat in the cloud today. More than half of the organizations currently use multiple clouds [20]. This considerable shift towards cloud-based operation introduces new techniques such as adopting microservice software to bring more flexibility and agility to NFV architecture. This phenomenon brought the so-called Cloud-native Network Function (CNF) [21]. CNF re-designed network functions to become self-contained, transforming them into a container format. This inspiration introduces challenges for supporting new applications, such as identifying and steering traffic for different users or content. Furthermore, cloud-native environments such as Kubernetes do not support NFV networking requirements, such as network isolation and fixed containers IP [31]. Lacking the support of network requirements endangers an essential task in NFV, Service Function Chaining (SFC). SFC is a mechanism that allows multiple different service functions to be connected to form a chain enabling carriers to benefit from the virtualized software-defined infrastructure. Hence, SFC is essential to spawning on-demand network services. In this paper, we concentrate on the design and implementation of SFC in a cloud-native environment, an approach that has not been addressed yet. While traditional SFC solutions have been widely addressed using the Software-Defined Networking (SDN) approach, SFC cloud-native is still somewhat novel in the research community. Our primary goal in this paper is detailing the design and the implementation of SFC concept deployed in a Kubernetes-based solution by leveraging Network Service Mesh (NSM) framework. The contribution of this paper is threefold:

1. We investigate various tools that can be utilized to build an SFC over Kubernetes
2. Inspired by the Network Service Mesh (NSM) approach, we design and implement a multi-node cloud-native SFC framework using a custom-build container in a Kubernetes cluster.
3. Performance analysis and evaluation of various Kubernetes SFC tools

The rest of the paper is organized as follows. Section 2 provides background on key concepts such as NFV, SFC, Kubernetes, containers, Container Network Interface (CNI), NSM, and the motivation behind our work. Section 3 describes our cloud-native traffic steering design. Section 4 provides details about the implementation process. In Sect. 5 we provide performance evaluation, while in Sect. 6, we provide the limitations and future work. Section 7 summarizes the main related work dealing with SFC in Kubernetes, and finally, the last section will conclude the paper.

2 Background

The main idea of NFV is to integrate proprietary network devices into industry-standard high-capacity servers, storage, and switches in various locations such as data centers [2]. The NFV architecture inherently promises advantages such as software and hardware decoupling, flexible network function deployment, and

dynamic functioning [3]. The Internet Engineering Task Force (IETF) describes Service Function Chaining (SFC) as an ordered set of service functions in which they must be executed [4]. Therefore, an SFC would consist of physical or virtualized network functions chained together where traffic will pass through before reaching the destination.

2.1 Motivation

SFC plays a vital role in next-generation networks by benefiting different technologies such as 5G, IoT, and edge computing [9, 10, 39, 40]. SFC helps in providing customizable network function services to traffic flows between different networks. The demand for new network services has grown exponentially, aided by the explosion of new network technologies and infrastructures, such as the success of cloud networks, that have increased the degree of pervasiveness and connectivity between heterogeneous devices. Our motivation behind this paper is to provide the reader with sufficient details regarding the design and implementation of SFC in Kubernetes using the NSM framework while delivering a real-life use case. This approach has not been addressed before in a cloud-native environment tool like Kubernetes. Focusing on chaining containers rather than traditional virtual machines, we created a cluster in Kubernetes that consisted of multiple Pods chained together and added video reduction and video broadcast service functions.

2.2 Kubernetes

Kubernetes is a cloud-based platform that offers a Container-as-a-Service (CaaS) layer for managing containerized workloads and services. According to Kubernetes' main web page, Kubernetes is “an open-source system for automating deployment, scaling, and management of containerized applications” [5]. The simplest way to describe Kubernetes' primary function is container cluster orchestration. Google originally designed Kubernetes, but since 2014 Cloud Native Computing Foundation (CNCF) has been responsible for maintaining Kubernetes. Kubernetes is a decentralized architecture based on a declarative model that defines the ultimate state. Users only describe the application's structure to be deployed when using the declarative model. In contrast, in the imperative model, the user must clearly define all the technical deployment tasks to be performed in sequence order. Kubernetes implement a microservice architecture that considers a single application as a collection of small services, each running in its process and communicating with each other with a lightweight mechanism, typically an HTTP service/gRPC service and corresponding API.

A cluster in Kubernetes consists of master and worker nodes. A node in Kubernetes may be either a virtual or a physical machine. Figure 1 is an example of a Kubernetes cluster that consists of two master nodes and three worker nodes. The main elements in a master node are different than those in a worker node. Intuitively from its name, the worker node performs the actual workload. Applications will have to be containerized, and a Pod will encapsulate the

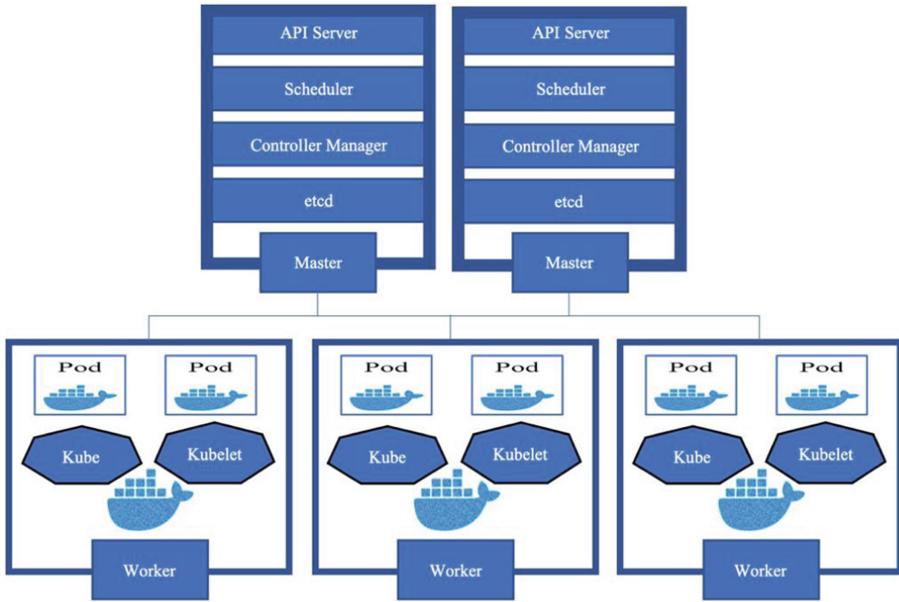


Fig. 1. Kubernetes cluster architecture and the main components.

container. The worker node will have at least one Pod. A Pod is the smallest deployable unit of computing that can be created and managed in Kubernetes. A Pod is a group of one or more containers in which they share resources such as shared storage, networking and information about how to run each container. Pods are temporary but can be automatically re-created to meet the desired state. There are three main background services running in each worker node. The first process is the container runtime which is responsible for running containers. Secondly, the Kubelet agent is the frontend CLI used to communicate with kube-apiserver, which resides on the master node. Thirdly, the Kube Proxy is a network proxy to forward the requests between different elements in the cluster.

A master node is considered a control plane that includes four main processes. The first process is API Server which acts as a cluster gateway and gatekeeper for authentication. All requests initiated by users will first pass through API Server for validation. The second process is Scheduler which is responsible for scheduling Pods on nodes while considering the resources in the cluster. The third process is the Controller Manager which is responsible for detecting cluster changes and bringing the cluster status to its desired state. The last process is etcd which is a key-value store for the cluster state [5].

Kubernetes offers minimal networking services. Kubernetes only provides the networking model placeholder and does not provide networking services/extensions in the cluster. In most cases, Kubernetes depend on third-party projects that provide network functionality. Four different types of network-

ing communication should be addressed when developing a network extension. Firstly, highly-coupled container-to-container communications. Secondly, Pod-to-Pod communications. Thirdly, Pod-to-Service communications and finally is the External-to-Service communications. A Service in Kubernetes is an abstract way to expose an application running on a set of Pods as a network service. Since Pods are ephemeral, Pods are associated with Services through key-value pair, where the Service will automatically discover new Pods with labels that match the key-value pair. Furthermore, Kubernetes imposes fundamental requirements on any networking implementation to allow Pods on a node to communicate with all Pods on all nodes without Network Address Translation (NAT). Third-party projects develop networking extensions that meet the networking module requirements, while each may have a different focus. Hence, users will have to choose amongst the available networking extensions that meet their needs to deploy them in Kubernetes.

2.3 Containers

Self-contained network functions are moved into a container, such as routers or firewalls. With containers, users can pack up their services neatly, including all application binaries, software dependencies, and necessary configuration files. This also means that the application will remain constant regardless of where they are running. Containers incur significantly lower overhead than traditional Virtual Machines (VMs). It is essential to mention that not all virtual network functions are feasible to be containerized [7]. Containers help businesses modernize by making it easier to scale and deploy applications. According to a CNCF survey done in 2020, containers usage in production has increased 300% since 2016, and 92% of users surveyed say they use containers [6]. Lightweight virtualization technologies such as cloud-native containers are the trend in deploying applications in the cloud infrastructure. Container-native Network Function (CNF) is a software implementation of a network function built and deployed in a cloud-native method [8]. Despite all the benefits gained from integrating containers into the NFV environment, there will be management and orchestration challenges that may hinder the utilization of container-based VNFs. Containers introduce new challenges and complexity by introducing an entirely new infrastructure ecosystem.

2.4 Different CNI Plug-ins

Networking in Kubernetes is provided by the so-called Container Network Interface (CNI), a CNCF project that defines the configuration of network interfaces for Linux containers. CNI comprises specifications and libraries for plug-ins to configure network interfaces in Linux containers [11]. A unique file called the CNI plug-in is responsible for inserting the correct network interface into the container network while making any necessary changes on the host. There are different kinds of CNIs, and each one provides a particular behaviour to allow networking inside the cluster. Some of the most common CNIs are Calico and

Canal, according to [12]. Firstly, Calico is well known for its performance, flexibility, and power. Calico provides additional functions, such as network security and administration, and essential Pod to Pod connections [13]. Secondly, Canal integrates Calico and another CNI called Flannel into one CNI to deploy in a Kubernetes cluster. It uses Flannel for networking pod traffic between hosts via VXLAN and Calico for network policy enforcement and Pod to Pod traffic [14]. Weave Net is another CNI plug-in. It is resilient and straightforward to use the network for Kubernetes and its hosted applications [42]. Weave Net creates a virtual network that connects Docker containers across multiple hosts and enables their automatic discovery. One of Weave Net's benefits is that it comes with a Network Policy Controller that automatically monitors Kubernetes for any NetworkPolicy annotations on all namespaces and configures iptables rules to allow or block traffic as directed by the policy

2.5 Network Service Mesh (NSM)

Kubernetes' principle includes service discovery and load balancing in an automated function for scaling up or down applications. On this basis, Kubernetes does not focus on the networking aspect but on managing a cluster. Kubernetes cannot provide some advanced L2/L3 network features, and it lacks the support for cross-cluster connectivity. Network Service Mesh (NSM) utilizes Kubernetes' networking model to perform specific networking functions. NSM is a novel approach to solving complicated L2/L3 use cases in Kubernetes that are tricky to solve [17], such as SFC use case. NSM is inspired by Software Defined Networking (SDN), in which NSM maintains the separation between control and data plane while providing network intelligence between microservices.

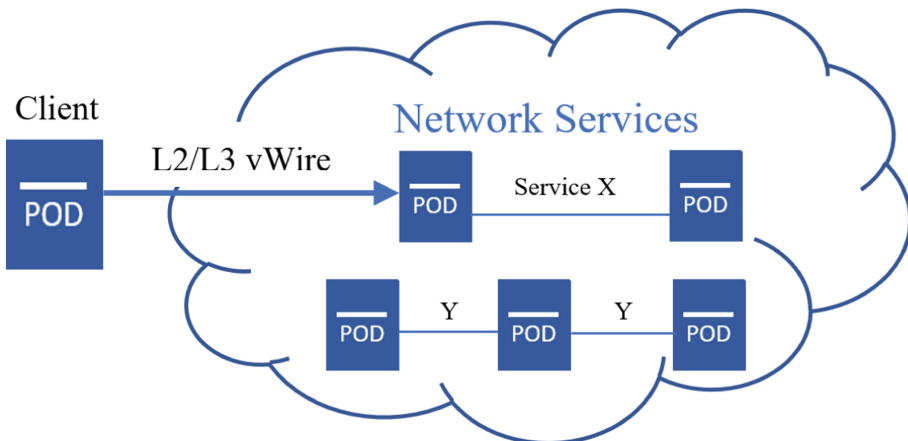


Fig. 2. NSM

NSM is based on three basic concepts. The first concept is Network Service (NS) which provides L2/L3 service. The second concept is Network Service Endpoint (NSE), a Pod in a Kubernetes cluster that provides the NS application. The final concept is the L2/L3 connection between the client's Pod and the NSE(s). NSM extends beyond kernel interface to support complex use cases and provides other interfaces such as memif or vhost-user interfaces. A memif interface, called Shared Memory Packet Interfaces, provides high-performance packet transmit and receive between the user application and Vector Packet Processing (VPP) [41]. NSM allows individuals to connect to an NS independently of the infrastructure they are running on. An NS, such as a chain of microservices, must be identified in a cluster to allow users to access it. After creating an NS, users will request to join a specific NS in the cluster by assigning a Pod to the user and creating a vWire to connect to the NS. User's Pod will have a unique annotation key-value pair that will specify which NS to connect. The NSM Manager will create a vWire that connects the user's Pod to the specified NS. A simple example of NSM connectivity is in Fig. 2. This figure illustrates how a client can access an NS on the Kubernetes cluster. Each NSE (Pod) includes a key-value pair to identify which NS it belongs to. In Fig. 2, a client would like to connect to Service X by sending a request to the NSM Manager. In return, the NSM Manager will examine the annotation key-value pair in the client's Pod and then check if the required NSEs and interface mechanism are available in the cluster. If there is a match, the NSM manager will respond to the request by creating a vWire to the appropriate requested NS.

NSM consists of a few elements that are important for its functionality. Figure 3 provides a graphical representation of the NSM control and data plane elements.

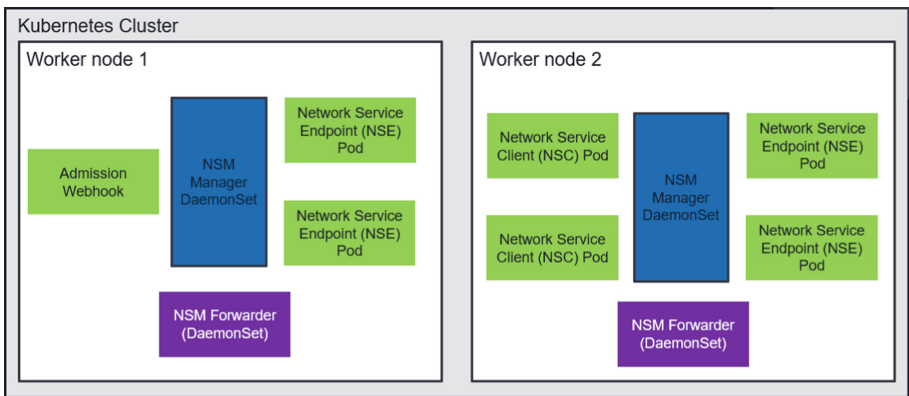


Fig. 3. NSM

Network Service Client is the first element involved in NSM. It is deployed as a Pod in a Kubernetes environment, and its main aim is to require a cross-

connection to a specific NS. On the other hand, NSE oversees implementing network functions in a network service. Both Network Service Client and NSE can be composed of two containers, one container for implementing NSM control plane functionalities and the other container implementing the primary service function. NSM Manager Pod is fundamental in the control plane that consists of three essential containers. The first container, which is the heart of the NSM control plane implementation, is the nsmd container. This container is responsible for all requests that involve cross-connections construction. The second container is nsmd-k8s, responsible for registries between different NSM Managers. The final container is nsmdp container which is in charge of checking that all elements involved in NSM Manager functions are working correctly. Network Service Forwarder's main aim is to implement NSM data-plane functionalities. When communication between Pods is provided, it is in charge of configuring interfaces and building cross-connection between involved Pods. Lastly, Admission Webhook intercepts Pod creation request to api-server and based on its internal configuration, and it can modify the request and inject specific code in the YAML request.

3 Design

In this paper, we address the problem of SFC in microservices-based architecture. Our contribution aims to provide details of the design and implementation process in deploying SFC in the Kubernetes cluster using custom-build containers by leveraging the NSM plug-in while adding extra features. Deploying an SFC in Kubernetes includes three steps: online search for third-party networking extension supporting SFC, deploy correct configurations for creating SFC and building an SFC in Kubernetes.

First Step is to search online for third-party projects (network extension) that support SFC in Kubernetes. Luckily, there are few options available. The first network extension is Contiv-VPP [15] which uses FD.io VPP to provide network connectivity between Pods in a Kubernetes cluster. The FD.io [16] is the world's secure networking data plane project that focuses on supporting terabit software data plane by using the VPP concept, which processes multiple packets at a time with low latency. Contiv-VPP is a CNI plug-in that employs a programmable CNF vSwitch offering SFC and other high-performance cloud-native networking and services. The second network extension is called OVN4NFV-K8s [19], and it is based on an Open Virtual Network (OVN) CNI controller to provide cloud-native-based SFC and other overlay networking features. OVN4NFV-K8s is a project under the Open Platform for NFV (OPNFV), a collaborative open-source platform for NFV. The third and final network extension that supports SFC in Kubernetes is NSM.

We tested the abovementioned three networking extensions to deploy SFC. We were able to deploy an SFC in a Kubernetes cluster using the Contiv-VPP extension successfully. Contiv-VPP provides three different scenarios to deploy

an SFC [45]. The first scenario is adding a tap interface to Linux CNFs. Secondly, each CNF Pod runs its own VPP instance and is connected with one or two additional memif interfaces. The final scenario is connecting a CNF to external Data Plane Development Kit (DPDK) sub-interfaces via two additional memif interfaces. The additional tap/memif interfaces between Pods/external interfaces are inter-connected on the L2 layer, using an L2 cross-connect on the vSwitch VPP. Contiv-VPP may be used on bare metal servers or using VMs. We went with VMs, where Kubernetes and service functions were on different VMs deployed on a single server.

We faced challenges in deploying SFC in OVN4NVF. The main issue was because the coreDNS Service Pod does not initiate. In other words, the API server could not get the endpoint of kube-dns Service. We ensured that no firewall was stopping the traffic and that coreDNS and API configurations were correct and functioning. OVN4NVF provides instructions on how to set up Kubernetes using VMs.

The third extension we tested was NSM, which is entirely orthogonal to standard Kubernetes networking. NSM allows Pods network with different workloads across the cluster using a simple set of APIs designed to provide connectivity, security, and observability. NSM leverages the Custom Resource Definition (CRD) service Kubernetes provides to define a custom resource in a cluster that performs a specific function [18]. NSM introduces an NS CRD, representing the logical implementation of a chain of network functions implemented as Pods in the cluster. The NS also specifies the order of the network function chain in which traffic should follow when traversing. It is also important to mention that the NSM control plane implements a cross-connection between Pods to allow proper communication. The cross-connection comprises two interfaces injected in the Pods involved in the communication.

Second Step is divided into two phases where the first phase is to deploy a cluster in Kubernetes, and the second phase is to configure the cluster according to the networking extension you choose in step one. Deploying a cluster in Kubernetes can be done using different tools. Kubeadm is a tool to build a Kubernetes cluster on a bare metal server [38]. Kubeadm toolbox will bootstrap a minimum viable Kubernetes cluster that conforms to best practice, allowing adding many nodes to the cluster. Another tool to build a cluster is using Kind tool. Kind is an open-source tool that generates Kubernetes clusters using Docker [25]. Kind was primarily designed for testing Kubernetes itself. Kind makes it easy to create a cluster by simply passing the command ‘kind create.’ NSM (release v0.2.0) uses the Kind tool to create clusters by default. Hence, Kind uses Kindnet [26] as the default networking plug-in. Kindnet implements the Kubernetes networking model using the CNI reference plug-ins and uses Docker’s default bridge networking. We created a cluster in Kubernetes that consisted of multiple Pods and services.

The second phase configures the Kubernetes cluster according to the network extension deployed to build an SFC. In general, all networking extensions follow

the same concept to identify an SFC service. Differences are mainly founded in the attribute values of the configuration files. The central idea is to create a CDR and reference the CRD in the services deployed in the cluster. The CRD is used to define an SFC with a name and schema. Figure 4a is a YAML configuration file of our custom CRD based on the NSM framework schema. It identifies a new custom resource that defines the concept of a networking service chain with the name of NetworkServiceChain, which will be used to create a network service chain, as Fig. 4b shows. Figure 4b is a YAML configuration file that identifies a chain of network services. We used the NetworkServiceChain name as an identifier and added it to the ‘kind’ attribute.

Traffic must follow the two matching rules, as depicted in Fig. 4b. Specifically, the first matching rule requires that the forwarder direct all traffic flow from any client that connects with this NS to the ‘firewall’ Pod, the entry point to the chain in the cluster. The matching rule is indicated in the red box in Fig. 4b. The second matching rule requires that traffic flow from the firewall Pod be steered to a second Pod, the ‘vid-reduction’ Pod, as indicated in the orange box in Fig. 4b. Hence, a flow request coming from an NSC Pod will be first headed to the ‘firewall’ Pod then to the ‘vid-reduction’ Pod. The metadata name attribute, ‘SFC-1’, in Fig. 4b is an essential attribute. This is the only method for attaching a Pod to a chain by having the metadata name of the chain in the Pod’s deployment configuration file.

Third Step in building an SFC in Kubernetes is to deploy the Pods in the cluster. This step is container development and adding them to a Pod. Developers need to create containers that will perform their application’s service/network function. After that, containers will be wrapped by Pods in the Kubernetes cluster. Our chain consists of three Pods in a sequence plus an extra Pod for the client, as illustrated in Fig. 5. Inside each Pod, we added a container that we custom-built to perform a specific function:

1. Firewall Pod: a firewall container to detect IP addresses and port numbers.
2. Vid-Reduction Pod: a container that performs video reduction size function.
3. Vid-Broadcast Pod: a container that broadcasts the video to users.

Docker is an open-source platform for building, deploying and managing containerized applications [22]. We used Docker to develop our containers and specifically included a CNI responsible for allocating network interfaces to the newly created container network namespace and making necessary changes on the host to enable the connectivity with other containers on the same network. Using the specification provided by the CNI GitHub [23], an IP address should be assigned to the interface using the correct IP address management.

Network Automation. To complete the design process, it is a good idea to automate the deployment process. There are multiple methods for automation, and one of the methods is to develop coding scripts ready for deployment. In our experiment, we created numerous Python scripts that configure, manage,

```

apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: networkservices.networkservicemesh.io
spec:
  conversion:
    strategy: None
  group: networkservicemesh.io
  names:
    kind: NetworkServiceChain
    listKind: NetworkServiceChainList
    plural: networkservicechains
    shortNames:
      - netsvcch
      - netsvcchs
    singular: networkservicechain
    scope: Namespaced

```

(a) CRD

```

apiVersion: networkservicemesh.io/v1
kind: NetworkServiceChain
metadata:
  name: SFC-1
  namespace: nsm-system
spec:
  payload: ETHERNET
  matches:
    - source_selector:
        app: firewall
      routes:
        - destination_selector:
            app: vid-reduction
    - routes:
        - destination_selector:
            app: firewall

```

(b) Network Service Chain

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: firewall
  annotations:
    ns.networkservicemesh.io: SFC-1
spec:
  template:
    spec:
      containers:

```

(c) Annotation Method

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: firewall
  annotations:
    ns.networkservicemesh.io: NetworkServiceChain
spec:
  template:
    spec:
      containers:
        - name: nsc-firewall
          env:
            - name: NSM_NETWORK_SERVICE
              value: kernel:///SFC-1/nsm-1
            - name: NSM_REQUEST_TIMEOUT
              value: 75s
      nodeSelector:
        kubernetes.io/hostname: ubuntu

```

(d) Environment Variables Method

Fig. 4. Kubernetes configuration files, YAML

and deploy the SFC in the Kubernetes cluster. We also built a simple web page, User Interface (UI), for clients to interact with the cluster to choose a video file from the available list for broadcasting. A request would be sent from the web page (frontend) to the backend to broadcast the requested video. The web page and the backend were developed using Python because of their simplicity in integrating the frontend to the backend process.

The first action the backend performs is creating a Pod for the client. Pod creation is crucial because the newly created Pod will contain metadata to identify which service function chain they would like to connect. There are two different methods to define the service function chain name in the NSM framework. Users can use annotation or include a variable in the environment specifications. The client's Pod configuration file will consist of an attribute called annotation, which specifies the name of the NS or the service function chain they would like

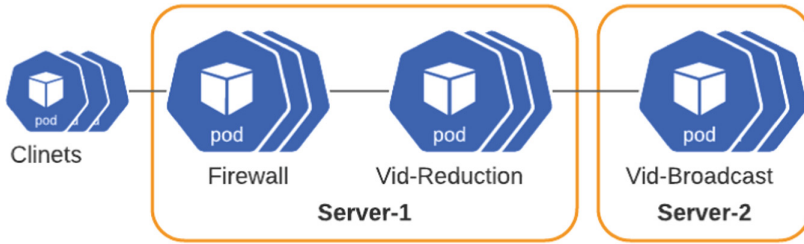


Fig. 5. SFC in Kubernetes multidomain cluster topology

to be associated with, as illustrated in Fig. 4c. The other method is to include a key value when deploying a Pod. The Pod will have an environmental value that is the exact value of the metadata of the network service chain, i.e. ‘SFC-1’, as illustrated in Fig. 4d. After creating the client’s Pod with the correct annotation or environment variable, a request would be sent to the Network Service Manager for connecting the client’s Pod to the specified NS. In return, the Network Service Manager will register the client as an NSC and search in the Network Service Registry for NSE. If an NSE is found in the registry, an interface will be injected into the client and related NSE Pods to create a chain. This chain will include only interconnected Pods, and each Pod will have separate NSM interface(s) where the Pod can communicate with other Pods in the chain.

The chain we developed includes three different service functions. The first service function (Pod) in the chain is the Firewall Pod. This Pod will act as an entry point to the chain in the cluster. Figure 4b illustrates this action. This Pod includes our custom-built container using the Nginx as a base image, and the primary function is to authenticate requests entering the Kubernetes cluster. If the request is allowed to enter the cluster, the traffic will be steered according to the chain identified, as illustrated in Fig. 4b, and the next hop in the chain will be the Vid-Reduction Pod. All traffic that egress the Firewall Pod will traverse to the Vid-Reduction Pod.

The second Pod in the chain is the Vid-Reduction Pod. Traffic from the previous Pod in the chain will ingress into this Pod which is responsible for locating the video file and checking the file size. If the file size is below a threshold, the file will be sent to the next hop in the chain without any modification. Otherwise, the video will be compressed. The Vid-Reduction Pod includes our custom-built container, which was developed using an Nginx base image. Furthermore, we use the FFmpeg tool to perform the compression function for the video file. After the compression function, the compressed file will be sent to the next hop in the chain. The file will leave from the Vid-Reduction Pod and traverse to the next hop in the chain, the Vid-Broadcast Pod.

The third and final Pod in the chain is the Vid-Broadcast Pod. Traffic from the previous Pod, Vid-Reduction Pod, in the chain will ingress into the Pod. This Pod is responsible for broadcasting the requested video to fulfill the client’s request. The Vid-Broadcast Pod container wraps the Nginx RTMP module and

FFMPEG tool. Nginx is open-source software for web servers, reverse proxying, load balancing, streaming, and more [24]. We choose Nginx because it provides a fast and reliable static web server, plus it is one of the most popular web servers.

4 Implementation

Testbed. The experiments were conducted on our testbed, CINE, consisting of 2 servers. One of the servers has a 40-core CPU (Intel Xeon E5-2650 @ 2.30) with one 1GbE NIC (Intel I350), and the other server has a 40-core CPU (Intel Xeon Silver 4114 @ 2.20GHz) with one 1GbE NIC (BRM 5720).

Kubernetes. We focused on using the latest version of Kubernetes. The client version for Kubernetes is 1.21.3, and the server version for Kubernetes is 1.21.1. We deployed Kubernetes on the two servers mentioned above. We also installed kubelet and kubectl. The kubelet is the component that runs on all the machines in the cluster and performs user's requests such as starting a Pod and containers. The latter is the command line to communicate with the cluster. We used the Kubectl tool to create clusters in Kubernetes.

Networking. We choose the NSM to be deployed in our cluster. The new release of NSM (v1.0.0) does not depend on a specific CNI. Therefore, we select Net Weave, a resilient and straightforward network for Kubernetes and its hosted applications [42]. Weave Net creates a virtual network that connects Docker containers across multiple hosts and enables their automatic discovery.

NSM. We tested two versions of the NSM releases. At first, we worked with the v0.2.0 release. This release was only released against Helm version 2. NSM is released through a set of Helm charts, which are easily deployable in the Kubernetes cluster. Release v0.2.0 introduces more features such as interdomain, DNS, security, and improvement to Network Service Endpoint. We also worked on the new release, v1.0.0, which was not officially published when we wrote the paper. Instead, the NSM community is releasing the latest version in phases. We worked with the new release and created a cluster with our custom-built containers to develop a chain of services. Release v1.0.0 added more features and capabilities from the old release, such as supporting different types of payloads (IP and Ethernet), latency reduction, and topology-aware scale.

5 Performance Evaluation

This section will provide a performance evaluation for the three different frameworks under three categories: Operating System (OS)-level virtualization, technology-based aspects, and management flexibility.

There are two different types of OS-level virtualization. The first one is VMs and the second is containers. The difference between VMs and containers is

the level of OS virtualization. Traditional VMs are heavyweight that run guest operating systems with their binaries, libraries, and applications that it services and the VM may be many gigabytes in size. In comparison, containers incur significantly lower overhead than traditional VMs and are gaining increasing attention in recent years [43]. A container shares host OS kernel, binaries and libraries, and they come in megabytes in sizes. Both OVN4NFV-K8s and Contiv-VPP use the VMs technique, which adds management overheads. Developers will have to deal with any additional issues when creating the VMs. While on the other hand, the NSM framework focuses on containers to reduce management overhead because they use the operating system's standard system call interface. But this comes with a flexibility issue where containers are not as flexible as a VM.

Secondly, each framework uses a different technology to present its solution for SFC. The Contiv-VPP extension only supports L2 cross-connect for interconnecting between Pods and only supports one single data path. Contiv-VPP relies on Data Plane Development Kit (DPDK) technology which offloads packet processing from the operating system kernel to userspace. Using DPDK technology brings benefits such as accelerating packet processing workloads. Despite that, it might be challenging to set up the correct environment and install DPDK for Contiv-VPP to function correctly on bare metal servers. On the other side, if you choose to implement Contiv-VPP using VMs, this will eliminate the challenges of installing DPDK as the VMs will be ready to use. Furthermore, Contiv-VPP requires a specific hypervisor, VirtualBox, limiting the users due to lack of support to VirtualBox hypervisor. It is essential to mention that Contiv-VPP only uses memif interfaces. Finally, Contiv-VPP provides a user interface that might help visualize the components and connect them.

OVN4NFV-K8s is based on Open Virtual Network (OVN), which supports virtual network abstraction and complements the existing capabilities of Open vSwitch that provides L2/L3 virtual networking, such as logical switches and routers, multiple tunnel overlays, and L2/L3/L4 ACLs. It is essential to mention that the OVN4NFV-K8s plug-in is a project under the Open Platform for NFV (OPNFV). Hence, it inherits and is limited to the OPNFV features. The third framework, NSM, complements traditional service mesh [44] and provides an infrastructure layer over microservices to standardize the runtime operations of applications. NSM focuses on supporting applications that might consist of many microservices, leading to simplicity, flexibility, and scalability. However, managing different microservices is a complex task, where different languages might be implemented, owned by different tenants, and/or constant changing states to microservices. Finally, comparing NSM with Contiv-VPP and OVN4NFV networking tools, NSM does not alter the Kubernetes CNI; instead, it is a standalone mechanism that consists of several components that can be deployed in a Kubernetes cluster. NSM provides different types of interfaces to be injected in a Pod. It gives the users a choice between using a memif or kernel interface.

The final category in evaluating the performance is each framework's flexibility for developers to configure the framework accordingly. Creating a Kubernetes

cluster using the OVN4NFV-K8s and Contiv-VPP framework was more complex than the NSM framework. Both OVN4NFV-K8s and Contiv-VPP require heavy pre-configuration. For OVN4NFV-K8s, the framework requires specific pre-configuration before deploying the cluster. Plus, it depends on building VMs rather than containers. On the other hand, deploying a Kubernetes cluster using NSM is smooth. We started by using Calico and implemented Calico as CNI with a single node cluster in using NSM. However, Calico causes some issues when switching to two physical node clusters. It delays connections between spire agent and spire server, consequently generating the failure of workload registration in NSM infrastructure. But NSM provides the freedom for users to choose amongst different networking plug-ins. Furthermore, NSM does not require heavy pre-configuration to deploy the networking plug-ins as it provides great flexibility.

6 Limitations and Future Work

We faced multiple challenges during our work to deploy an SFC in Kubernetes. The first challenge was working with NSM releases. Between old and new releases of the NSM framework, the documentation provided is inferior. The latest release of NSM introduces a new method to deploy Network Service. It involves the Kubernetes concepts of the Kustomize tool, a standalone tool to customize Kubernetes objects through a kustomization file [27]. We faced the second challenge of integrating our custom-built containers into the NSM framework. Specifically when adding the feature of injecting memif interfaces to coexists with our service function. SM framework forces the injection of interfaces, and traffic will have to ingress and egress specifically from those interfaces. This can limit service functions types implemented in a chain. The final challenge we faced was the transfer process of the video file between Pods. Transferring files between Pods in a chain is different than regular file transfer using Kubernetes-based features. There are many tools for file transfer, such as the secure copy protocol, but implementing it in a container will increase the container image size. This will eliminate one of the container's benefits of being lightweight.

SFC is still not mature in microservice-based network architecture. More research is needed to provide solutions for chaining service functions while using containers and not traditional VMs. Furthermore, the SFC concept is limited to small applications, such as load balancing and packet investigations. Big science data flow applications might benefit from SFC features if deployed correctly. Another area of improvement is an analytical study of the effect of different network interfaces performance. In our example, we used both kernel and memif interfaces. Ideally, it would be good to analyze how each different interface performs in a container environment. Finally, applying network analysis to extract network metrics and optimize the performance will provide a better QoS over the chain.

7 Related Work

The authors in [28] provide a similar work by using NSM in Kubernetes to offer SFC solutions. They proposed an efficient traffic steering orientation for cloud-native service function chaining. They proposed a new network-aware traffic orientation model based on weighted cycles. This is different from our work as we focus on SFC's design and implementation process using the NSM framework in Kubernetes. Also, in [29], they offered a solution to maximize the QoS satisfaction rate by load-balancing the traffic over the SFC path using the Convtiv-VPP method. Few papers [30,31] focus on integrating OpenStack and Kubernetes to deploy a chain of service functions. OpenStack provides VMs for users to deploy their services and applications, while Kubernetes orchestrates and manages containers. Bringing both OpenStack and Kubernetes together uses Kuryr, an OpenStack project that aims to solve container networking issues in OpenStack. Many papers fill in the gap for container-based orchestration. Since there is no standard for defining container-based VNFs, many articles fill the gap by designing new solutions such as extending Tacker architecture (NFV management and orchestration framework) [32,33]. The authors in [34] proposed a fault management system with dynamic policy recovery enforcement to support the high availability of SFC in a multi-cloud environment. In [35], the authors proposed a performance model approach for recommending an initial resource provisioning for every microservice within all CNFs before deploying the SFC. Another interesting paper [36] proposed a machine learning framework module that can detect anomalies for SFC integrity. Finally, a recent paper [37] proposed a resource and energy-aware SFC strategy in the edge-cloud environment for IoT applications that would cope with dynamic load and resource situations emerging from dynamic SFC requests. Our work is related to those papers mentioned in this section by building a service function chain. We took a different avenue by providing sufficient details on SFC's design and implementation process using the NSM framework in Kubernetes and adding more value to a service function chain. To our knowledge, no previous work used the NSM framework for building a chain of network services using real-life use cases.

8 Conclusion

NFV is the future technology that enables cloud-based platforms to provide public services and acquire resources such as networking, computing, and storage. This concept unfolded innovations such as container-based microservices for deploying services and applications. Containers are efficient and flexible while incurring significantly lower overhead. Kubernetes is a tool to orchestrate and manage containers. Kubernetes' function strategy follows a declarative microservice approach. Kubernetes provides service discovery and load balancing, automation in self-healing, optimal scheduling, and security mechanisms. It also has a shorter time to deployment due to architecture, logging detail and live "in-service" debugging. Kubernetes does provide a specific way to interconnect

Pods and containers. Instead, it depends on the third party to provide overlay network functions such as NSM over Kubernetes essential network functions. These projects follow the Kubernetes networking model to build a networking plug-in for the Kubernetes cluster. We provide details on different networking extensions that support SFC. We also briefly explain the various networking plug-ins that support CNI, such as Calico, Canal, and Contiv-VPP.

We created a Kubernetes cluster using the NSM framework, supporting the SFC concept. We created a service function chain that consisted of multiple Pods in a multi-node cluster. The Pods contained our custom-built containers, and each container was built to perform a different function. The container functions we built were firewall, video compression, and video broadcasting containers. We found limitations when using the NSM framework to deploy the SFC concept. Hence, the limitations of deploying the SFC concept on Kubernetes are related to the functionality and features of the networking extension plug-in we used (NSM). Our SFC design and implementation focused on providing a real-life scenario compared to traditional chains with limited service functions.

Acknowledgement. This project is supported by the Mitacs Accelerate program funded by NSERC between Ciena and Carleton University, Ottawa, Canada.

References

1. Tsuji, Y., Itoh, A., Kobayashi, M.: Future network technologies for the 5G/IoT Era. *NTT Tech. Rev.* **16**(6) (2018)
2. ETSI Industry Specification Group (ISG): Network Functions Virtualisation (NFV): An introduction, benefits, enablers, challenges and call for action. SDN and OpenFlow World Congress, Darmstadt, Germany (2012)
3. ETSI Industry Specification Group (ISG): Network Functions Virtualisation (NFV): Architectural Framework (2014)
4. Halpern, J., Pignataro, C.: Service Function Chaining (SFC) Architecture. In: RFC, number 7665, October 2070–1721, RFC Editor, RFC Editor (2015)
5. Kubernetes, Production-Grade Container Orchestration, <https://kubernetes.io/>. Accessed 15 Nov 2021
6. Cloud Native Computing Foundation, CNCF Survey Report 2020. <https://www.cncf.io/wp-content/uploads/2020/12/CNCF-Survey-Report-2020.pdf>. Accessed 15 Nov 2021
7. Cziva, R., Pezaros, D.P.: Container network functions: bringing NFV to the network edge. *IEEE Commun. Mag.* **55**(6), 24–31 (2017). <https://doi.org/10.1109/MCOM.2017.1601039>
8. Cloud-Native Network Functions. <https://cdnf.io/>. Accessed 15 Nov 2021
9. Li, X., Rao, J., Zhang, H., Callard, A.: Network Slicing with Elastic SFC. In: IEEE 86th Vehicular Technology Conference (VTC-Fall), pp. 1–5 (2017). <https://doi.org/10.1109/VTCFall.2017.8287914>
10. Barakabitze, A.A., et al.: 5G network slicing using SDN and NFV: a survey of taxonomy, architectures and future challenges. *Comput. Netw.* **167**, 106984 (2020). <https://doi.org/10.1016/j.comnet.2019.106984>
11. CNI - the container network interface. <https://github.com/containernetworking/cni>. Accessed 15 Nov 2021

12. Benchmark-k8s-cni-2020-08. <https://github.com/InfraBuilder/benchmark-k8s-cni-2020-08>. Accessed 15 Nov 2021
13. Project Calico. <https://docs.projectcalico.org/getting-started/kubernetes/>. Accessed 15 Nov 2021
14. KOPS -Kubernetes Operation. <https://kops.sigs.k8s.io/networking/canal/>. Accessed 15 Nov 2021
15. Contivpp. <https://contivpp.io/>. Accessed 15 Nov 2021
16. FD.io, The world's secure networking data plane. <https://fd.io/>. Accessed 15 Nov 2021
17. Network Service Mesh. <https://networkservicemesh.io/>. Accessed 15 Nov 2021
18. Custom Resources. <https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/>. Accessed 15 Nov 2021
19. OPNFV/OVN4NFV-K8s-K8s-plugin. <https://github.com/opnfv/ovn4nfv-k8s-plugin/>. Accessed 15 Nov 2021
20. IDG 2020 IDG Cloud Computing Study. <https://resources.idg.com/download/2020-cloud-computing-executive-summary-rl/>. Accessed 15 Nov 2021
21. CDNf, Cloud-Native Network Functions. <https://cdnf.io>. Accessed 15 Nov 2021
22. Docker Homepage. <https://www.docker.com/>. Accessed 15 Nov 2021
23. Container Network Interface specification. <https://github.com/containernetworking/cni/blob/master/SPEC.md>. Accessed 15 Nov 2021
24. NGINX Homepage. <https://www.nginx.com/>. Accessed 15 Nov 2021
25. Kind Homepage. <https://kind.sigs.k8s.io/>. Accessed 15 Nov 2021
26. Simple CNI plugin with IPv4, IPv6 and DualStack support. <https://github.com/aojea/kindnet>. Accessed 15 Nov 2021
27. Declarative management of Kubernetes objects using kustomize. <https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization/>. Accessed 15 Nov 2021
28. Dab, B., Fajjari, I., Rohon, M., Auboin, C., Diquélou, A.: An efficient traffic steering for cloud-native service function chaining. In: 23rd conference on innovation in clouds, Internet and Networks and Workshops (ICIN), pp. 71–78 (2020). <https://doi.org/10.1109/ICIN48450.2020.9059340>
29. Bouridah, A., Fajjari, I., Aitsaadi, N., Belhadeif, H.: Optimized scalable SFC traffic steering scheme for cloud native based applications. In: IEEE 18th Annual Consumer Communications & Networking Conference (CCNC), pp. 1–6 (2021). <https://doi.org/10.1109/CCNC49032.2021.9369583>
30. Vu, X.T., et al.: An architecture for enabling VNF auto-scaling with flow migration. In: 2020 International Conference on Information and Communication Technology Convergence (ICTC), pp. 624–27. IEEE (2020). <https://doi.org/10.1109/ICTC49870.2020.9289507>
31. Kouchaksaraei, H.R., Karl, H.: Service function chaining across openstack and kubernetes domains. In: Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems (2019)
32. Hoang, C.-P., et al.: An extended virtual network functions manager architecture to support container. In: Proceedings of the 2018 International Conference on Information Science and System, pp. 173–176. ACM (2018). <https://doi.org/10.1145/3209914.3209934>
33. Yang, H., Hoang, C., Kim, Y.: Architecture for virtual network function's high availability in hybrid cloud infrastructure. In: 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), pp. 1–5 (2018). <https://doi.org/10.1109/NFV-SDN.2018.8725784>

34. Song, S.-Y., Lin, F.J.: Dynamic fault management in service function chaining. In: IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), pp. 1477–1482. IEEE (2020). <https://doi.org/10.1109/COMPSAC48688.2020.00-46>
35. Khan, M.G., et al.: A performance modelling approach for SLA-aware resource recommendation in cloud native network functions. In: 6th IEEE Conference on Network Softwarization (NetSoft), pp. 292–300 (2020). <https://doi.org/10.1109/NetSoft48620.2020.9165482>
36. Cheng, S.-T., Zhu, C.-Y., Hsu, C.-W., Shih, J.-S.: The anomaly detection mechanism using extreme learning machine for service function chaining. In: 2020 International Computer Symposium (ICS), pp. 310–315 (2020). <https://doi.org/10.1109/ICS51289.2020.00068>
37. Thanh, N.H., Kien, N.T., Van Hoa, N., Huong, T.T., Wamser, F., Hossfeld, T.: Energy-aware service function chain embedding in edge-cloud environments for IoT applications. *IEEE Internet Things J.* **8**(17), 13465–13486 (2021). <https://doi.org/10.1109/JIOT.2021.3064986>
38. Creating a cluster with kubeadm. <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/>. Accessed 15 Nov 2021
39. Zou, D., Huang, Z., Yuan, B., Chen, H., Jin, H.: Solving anomalies in NFV-SDN based service function chaining composition for IoT network. *IEEE Access* **6**, 62286–62295 (2018). <https://doi.org/10.1109/ACCESS.2018.2876314>
40. Imagane, K., Kanai, K., Katto, J., Tsuda, T., Nakazato, H.: Performance evaluations of multimedia service function chaining in edge clouds. In: 15th IEEE Annual Consumer Communications & Networking Conference (CCNC), pp. 1–4 (2018). <https://doi.org/10.1109/CCNC.2018.8319249>
41. Memif Poll Mode Driver. <https://doc.dpdk.org/guides/nics/memif.html>. Accessed 15 Nov 2021
42. Weaveworks, Integrating Kuberntes via the Addon. <https://www.weave.works/docs/net/latest/kubernetes/kube-addon/>. Accessed 15 Nov 2021
43. Zhang, Q., Liu, L., Pu, C., Dou, Q., Wu, L., Zhou, W.: A comparative study of containers and virtual machines in big data environment. In: IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 178–185 (2018). <https://doi.org/10.1109/CLOUD.2018.00030>
44. Li, W., Lemieux, Y., Gao, J., Zhao, Z., Han, Y.: Service mesh: challenges, state of the art, and future research opportunities. In: 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 122–1225 (2019). <https://doi.org/10.1109/SOSE.2019.00026>
45. CONTIV/VPP. <https://github.com/contiv/vpp/tree/master/k8s/examples/sfc>. Accessed 31 Jan 2022

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Author Index

- Aghasharif, Amir 121
Alpay, Aksel 79
- Bastrakov, Sergei 79
Bittar, Abdullah 121
Bussmann, Michael 79
- Chen, Zhong 20
Chiew, Shao-Hen 43
- Debus, Alexander 79
- Esposito, Aniello 79
- Gerofi, Balazs 59
- Heikonen, Jussi 79
Hori, Atsushi 59
Huang, Changcheng 121
- Imamura, Toshiyuki 1
Ishikawa, Yutaka 59
- Klemm, Michael 79
Kluge, Thomas 79
- Kopeć, Jakub 102
Kwek, Leong-Chuan 43
- Lee, Chee-Kong 43
Lyonnais, Marc 121
- Machida, Masahiko 1
Malaya, Nicholas 79
Markomanolis, George S. 79
- Ouyang, Kaiming 59
- Shami, Gauravdeep 121
Steiniger, Klaus 79
Stephan, Jan 79
- Wang, Pengyu 20
Wang, Ziqiang 121
Widera, Rene 79
Wilson, Rodney 121
- Yamada, Susumu 1
Yao, Jie 31
Yeo, K. S. 31
Young, Jeffrey 79