

Festschrift

LNCS 14564

Rajkishore Barik  
Rajiv Gupta  
Jens Palsberg (Eds.)

# Principles and Practices of Building Parallel Software

**Essays Dedicated to Vivek Sarkar  
on the Occasion of His 64th Birthday**



Springer

# Lecture Notes in Computer Science

14564

Founding Editors


Gerhard Goos


Juris Hartmanis

## 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*

The series Lecture Notes in Computer Science (LNCS), including its subseries Lecture Notes in Artificial Intelligence (LNAI) and Lecture Notes in Bioinformatics (LNBI), has established itself as a medium for the publication of new developments in computer science and information technology research, teaching, and education.

LNCS enjoys close cooperation with the computer science R & D community, the series counts many renowned academics among its volume editors and paper authors, and collaborates with prestigious societies. Its mission is to serve this international community by providing an invaluable service, mainly focused on the publication of conference and workshop proceedings and postproceedings. LNCS commenced publication in 1973.

Rajkishore Barik · Rajiv Gupta · Jens Palsberg  
Editors

# Principles and Practices of Building Parallel Software

Essays Dedicated to Vivek Sarkar  
on the Occasion of His 64th Birthday

*Editors*

Rajkishore Barik  
Gitar Inc.  
San Mateo, CA, USA

Rajiv Gupta  
University of California  
Riverside, CA, USA

Jens Palsberg  
University of California  
Los Angeles, CA, USA

ISSN 0302-9743

ISSN 1611-3349 (electronic)

Lecture Notes in Computer Science

ISBN 978-3-031-97491-5

ISBN 978-3-031-97492-2 (eBook)

<https://doi.org/10.1007/978-3-031-97492-2>

© The Editor(s) (if applicable) and The Author(s), under exclusive license  
to Springer Nature Switzerland AG 2025

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

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

If disposing of this product, please recycle the paper.



# Preface

It is our great pleasure to dedicate this Festschrift volume to the scholarship, leadership, and teaching of Professor Vivek Sarkar, the John P. Imlay, Jr. Dean of the College of Computing at Georgia Tech and a distinguished professor in the School of Computer Science.

The title of this Festschrift, *Principles and Practices of Building Parallel Software*, highlights Vivek's transformative contributions to advancing parallel computing. His work spans programming languages, compilers, runtime systems, debugging tools, and verification of programs, all designed to address the challenges of high-performance and exascale computing. Through pioneering innovations, technical expertise, and dedicated mentorship, Vivek's career has profoundly shaped both industry practices and academic research, establishing him as a role model for generations of computer scientists.

Vivek's journey began with his foundational Ph.D. work at Stanford University under the mentorship of Prof. John Hennessy, a luminary in computer science. In the 1980s, when parallel programming was still in its infancy, Vivek made significant advances in the scheduling of parallel programs, addressing key challenges in optimizing dependencies and laying the foundation for modern compiler optimizations that unlock parallelism at scale.

After completing his Ph.D., Vivek joined IBM Research, where, under the mentorship of Fran Allen, he contributed to the PTRAN Project by developing the PART partitioner for automatic parallelization, analyzing cost-benefit tradeoffs while accounting for overhead and synchronization costs. At IBM Santa Teresa Labs, he led the design and implementation of the ASTI optimizer for IBM's XL compiler, pioneering advanced program transformations such as loop distribution, tiling, and scalar replacement, seamlessly integrating cutting-edge compiler techniques into product.

Among Vivek's most influential contributions is the design of the X10 programming language, an object-oriented approach to improve the productivity of high-performance computing. His seminal paper, *X10: an Object-Oriented Approach to Non-Uniform Cluster Computing* [2], which introduced innovative programming abstractions for parallel and distributed systems, won the Most Influential Paper Award for OOPSLA 2005 and continues to shape research in scalable parallel programming.

Vivek also led the development of the Jikes Research Virtual Machine (RVM) [1, 3], an open-source JVM that enabled experimentation with advanced virtual machine technologies and influenced the evolution of managed runtime systems. It has been used by over 100 universities worldwide, serving as the foundation for more than 200 research publications, 40 doctoral dissertations, and 20 university-level courses. The project was honored with the prestigious SIGPLAN System Software Award in 2012.

In academia, Vivek has been a transformative leader, serving as Department Chair at Rice University and Georgia Tech, and now as Dean of the College of Computing at Georgia Tech. At Rice, he led the Habanero Extreme Scale Software Research group, which introduced novel runtime systems and programming models, such as task-parallel abstractions and data-driven synchronization primitives, that significantly advanced the productivity and scalability of extreme-scale parallel applications.



**Fig. 1.** IBM ASTI team photo with Prof. Vivek Sarkar.

Beyond his research, Vivek has played pivotal roles on advisory committees such as the US Department of Energy’s Advanced Scientific Computing Advisory Committee (ASCAC) and as co-chair of the CRA-Industry Committee. A recipient of the prestigious ACM-IEEE CS Ken Kennedy Award, his influence spans the global computing community, fostering collaboration between academia and industry.

It was heartwarming to witness the overwhelming response to VIVEKFEST, held on October 21, 2024, in Pasadena, California, as part of SPLASH’24. This symposium brought together many of Vivek’s collaborators, colleagues, current and former students, industrial fellows, and friends to celebrate his remarkable career. The day-long event featured an exceptional lineup of technical talks highlighting Vivek’s contributions to programming languages, compiler technologies, and runtime systems. These presentations were punctuated with personal anecdotes and references to Vivek’s work.

Attendees traveled from far and wide to honor Vivek’s legacy on October 21, 2024, underscoring the far-reaching impact of his work. The symposium also provided an opportunity to relax and socialize during the evening reception, where colleagues and friends shared their appreciation for Vivek’s mentorship and his tireless efforts to advance the field. The event was a fitting tribute to a career that has profoundly shaped computing research and education.





**Fig. 2.** Vivek with the Jikes Research Virtual Machine (RVM) team in 2001



**Fig. 3.** Research team led by Prof. Vivek Sarkar, pictured at Rice University

We, as organizers, are deeply thankful to the authors who contributed their research contributions to this volume and to the reviewers who provided invaluable feedback. Special thanks to Springer for publishing this Festschrift as part of their Lecture Notes in Computer Science series.

Beyond his scholarly contributions, Vivek's leadership and mentorship have inspired generations of computer scientists, fostering a culture of excellence and collaboration. A remarkable scholar and visionary, Vivek is also a great human being whose humility and warmth have touched countless lives. His dedication to his family reflects his values as a devoted husband and father, embodying the balance of professional achievement and personal fulfillment. As we celebrate his 64th birthday, we recognize that Vivek's journey is far from over and look forward to his continued contributions in addressing the challenges of exascale computing.

This Festschrift serves as a tribute to his extraordinary achievements and a source of inspiration for those who follow in his footsteps.



**Fig. 4.** Research team led by Prof. Vivek Sarkar, pictured at Georgia Tech



**Fig. 5.** Vivek Sarkar with his beloved wife, Ranta Sarkar

Happy 64th birthday, Vivek! Thank you for your lasting contributions, your mentorship, and your unwavering commitment to advancing the frontiers of computer science.

November 2024

Rajkishore Barik  
Rajiv Gupta  
Jens Palsberg

## References

1. Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In Proceedings of the ACM 1999 Conference on Java Grande, JAVA'99, page 129–141, New York, NY, USA, 1999. Association for Computing Machinery.
2. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. New York, NY, USA, 2005. Association for Computing Machinery.
3. IBM. Jikes Research Virtual Machine (RVM). <https://www.jikesrvm.org/>, 2005.

## Photos from the Symposium



**Fig. 6.** Attendees of the VIVEKFEST Festschrift Symposium (Photo by Madhurima Chakraborty)



**Fig. 7.** A memorable dinner with colleagues on the eve of the VIVEKFEST celebration





**Fig. 8.** VIVEKFEST kicks off with an opening address by Vivek's former doctoral student, Rajkishore Barik



**Fig. 9.** Prof. Vivek Sarkar delivering his inspiring speech



**Fig. 10.** Prof. V. Krishna Nandivada sharing his stories about Vivek



**Fig. 11.** Prof. Tiago Cogumbreiro sharing his stories about Vivek



**Fig. 12.** Prof. Ganesh Gopalakrishnan sharing his stories via pre-recorded video



**Fig. 13.** Prof. Zoran Budimlić sharing heartfelt stories through a pre-recorded video



## Personal Notes for Vivek

**From Evelyn Duesterwald** Vivek is such an inspiring role model, equal parts technical brilliance and genuine care for the people around him. I am still a little sad he left IBM so early, it's not the same without him.

**From John Richards** Vivek has made profound contributions to high productivity systems and X10.

**From Michael Hind** Vivek was like that great parent, you remember the things he did, but also later realize the things he didn't do. He allowed you to be independent, be innovative, and grow.

**From John Field** Heartfelt congratulations to Vivek on his 64th year of contributions to the world!

I had the great pleasure of working under Vivek at IBM Watson. There, he practiced the fine art of maintaining a protective shield over research projects in his domain while nudging new work in directions most likely to yield funding and recognition. The lessons I learned from Vivek played a major role in shaping my own approach to management at Google.

Vivek, I wish you many more years of influence and renown!

**From Mark Wegman** Vivek did have an important career at IBM. I think he learned some stuff here as well as contributing.

**From Jinfan Shaw** Thank you for your kind invitation to VivekFest in celebration of Vivek Sarkar's 64th birthday. It is with great pleasure that I learned of this event honoring a former colleague whose contributions to the ASTI project at IBM I hold in high esteem.

While I am unable to attend the event in person due to prior commitments, I would have been honored to participate and share my recollections of our collaboration on the ASTI project.

I first met Vivek in 1988 during a presentation on Parallel Fortran at Watson Research. A subsequent conversation during a leisurely post-lunch walk around the Hawthorn research center proved to be both stimulating and enjoyable.

In early 1990, Vivek joined the VS Fortran team at the Silicon Valley Lab to take the lead in designing and implementing the transformer component of the ASTI optimizer. Under his direction, the transformer was developed to perform a range of program transformations including loop distribution, interchange, reversal, skewing, tiling, fusion, unrolling, and scalar replacement of array references. The ASTI transformer was designed to select these optimizations automatically.

Leveraging his research background and connections at IBM Research, Vivek was instrumental in incorporating static single assignment (SSA) and Interprocedural Analysis into the ASTI optimizer. Subsequently, he collaborated with the Toronto compiler

group to integrate the ASTI optimizer into the IBM XLF compiler for RS/6000 and PowerPC systems.

I extend my warmest congratulations to Vivek on his birthday and look forward to hearing more about VivekFest.

# Organization

This Festschrift was organized by Vivek's former doctoral student Rajkishore Barik (Gitar Inc., USA), alongside his esteemed colleagues and dear friends Rajiv Gupta (University of California, Riverside, USA) and Jens Palsberg (University of California, Los Angeles, USA).

**Acknowledgments.** We sincerely thank Manu Sridharan for his invaluable guidance in organizing the event and for his support throughout its execution. We are grateful to Springer for their invaluable advice and support. Last but not least, we would like to acknowledge the reviewers for their constructive feedback.

## Reviewers

Samuel Pollard

Jun Shirako

Tiago Cogumbreiro

Oscar Hernandez

Jisheng Zhao

Akihiro Hayashi

Feiyang Jin

V. Krishna Nandivada

Louis-Noel Pouchet

Yonghong Yan

Sandia National Laboratories, USA

Georgia Institute of Technology, USA

University of Massachusetts, Boston, USA

Oak Ridge National Laboratory, USA

Georgia Institute of Technology, USA

Georgia Institute of Technology, USA

Georgia Institute of Technology, USA

Indian Institute of Technology, Madras, India

Colorado State University, USA

University of North Carolina at Charlotte, USA

# Contents

Retrieving Unknown SMT Formulas via Structural Mutations .....	1
<i>Shuo Ding and Qirun Zhang</i>	
On the Cloud We Can't Wait: Asynchronous Actors Perform Even Better on the Cloud .....	11
<i>Aniruddha Mysore, Youssef Elmougy, and Akihiro Hayashi</i>	
A Formal Model for Portable, Heterogeneous Accelerator Programming .....	22
<i>Zachary J. Sullivan and Samuel D. Pollard</i>	
Evaluation of Speedup and Energy with Multigrain Parallelizing Compiler .....	34
<i>John Pickar, Tohma Kawasumi, Hiroki Mikami, Keiji Kimura, and Hironori Kasahara</i>	
Concurrent Collections: An Overview .....	50
<i>Kathleen Knobe, Zoran Budimlić, Robert J. Harrison, Mohammad Mahdi Javanmard, and Louis-Noël Pouchet</i>	
Hidden Assumptions in Static Verification of Data-race Free GPU Programs ...	55
<i>Tiago Cogumbreiro and Julien Lange</i>	
Intrepydd: Toward Performance, Productivity, and Portability for Massive Heterogeneous Parallelism .....	64
<i>Jun Shirako, Tong Zhou, and Akihiro Hayashi</i>	
Enabling User-Level Asynchronous Tasking in the FA-BSP Model Case Study: Distributed Triangle Counting .....	70
<i>Akihiro Hayashi, Shubhendra Pal Singhal, Youssef Elmougy, and Jiawei Yang</i>	
Learning to Harness In-Vitro Biological Neural Networks .....	78
<i>Frithjof Gressmann and Lawrence Rauchwerger</i>	
Verification of Concurrent Programs Using Hybrid Concrete-Symbolic Interpretation .....	90
<i>Emily Tucker and Louis-Noël Pouchet</i>	
Scalable Small Message Aggregation on Modern Interconnects .....	103
<i>Aaron Welch, Oscar Hernandez, Stephen Poole, and Wendy Poole</i>	

Preliminary Study on Message Aggregation Optimizations for Energy  
Savings in PGAS Models ..... 114  
    *Oscar Hernandez, Aaron Welch, Wendy Poole, and Stephen Poole*

**Author Index ..... 121**



# Retrieving Unknown SMT Formulas via Structural Mutations

Shuo Ding<sup>✉</sup> and Qirun Zhang<sup>(✉)</sup><sup>✉</sup>

Georgia Institute of Technology, Atlanta, GA 30332, USA

sding@gatech.edu, qrzhang@gatech.edu

**Abstract.** Satisfiability Modulo Theories (SMT) solvers are fundamental tools for program analysis and verification. The satisfiability problem for first-order logic is undecidable. In practice, SMT solvers typically employ various heuristics and are inherently *incomplete*. Solvers return **unknown** if they cannot solve a particular formula. The **unknown** results drastically hinder the usability of SMT solvers and directly affect client applications. The standard way to reduce **unknown** cases is to develop more powerful solvers, which requires significant algorithmic and engineering efforts.

This work-in-progress paper discusses a new perspective on improving SMT solving: instead of developing more powerful solvers for all formulas, we focus on mutating “hard” formulas (**unknown** formulas) to make them “easier” to solve. That gives us enormous flexibility to process **unknown** formulas without affecting normal formulas. Specifically, given an **unknown** formula and a solver, we propose to repeatedly modify the formula via structural mutations. Our key insights are (1) structural mutations make formulas smaller so that they are presumably easier to reason about, and (2) structural mutations approximate formulas so that we can reason about the original formulas indirectly. Then, we utilize the same solver to solve the mutated formulas to retrieve the **sat/unsat** results of the original **unknown** formulas.

## 1 Introduction

Satisfiability Modulo Theories (SMT) is a powerful formulation that can express many problems arising in symbolic execution [12,30], formal verification [7,23], program synthesis [19], etc. An SMT problem instance describes a first-order logic formula with respect to certain background theories. SMT solvers are software tools for deciding the satisfiability of SMT formulas. Z3 [25] and CVC4 [5] (now succeeded by CVC5 [3]) are two widely used SMT solvers. However, it is well-known that the satisfiability problem for first-order logic is undecidable. In addition to theoretical restrictions, modern SMT solvers also face practical issues, including incomplete implementations and resource limits. Therefore, practical SMT solvers return **unknown** results for formulas that they cannot solve. In the popular Satisfiability Modulo Theories Competition (SMT-COMP), in

many tracks, a solver receives a zero “correctly solved score” if the `check-sat` command returns `unknown` [4]. In practice, SMT solvers strive to offer best-effort answers by solving as many formulas as possible.

The standard way to reduce `unknown` cases is by improving solvers, including developing new algorithms and engineering better solvers. That is challenging and time-consuming due to both the theoretical hardness of SMT solving and the implementation issues of such complex systems. For example, CVC4’s bitvector rewriting rules contain more than 3.5K source lines of code [32]. From users’ perspective, it is possible to try solvers’ available options or tactics, or even different solvers to handle `unknown` cases, but there are usually a limited number of choices at a given time.

We consider a source-level approach for improving SMT solving. Rather than developing more powerful solvers for all formulas, we focus on “hard” formulas for which solvers return `unknown`. Working directly on `unknown` formulas enables unique opportunities for employing solver-agnostic source-to-source transformations to make “hard” formulas easier to solve. Specifically, given an `unknown` formula  $\phi$  for a specific solver, we propose a technique called *structural mutations* to perform lightweight rewriting on  $\phi$  and obtain a mutated formula  $\phi'$ . Then we apply the same solver on  $\phi'$  to reason about  $\phi$  indirectly. There are two key observations that underlie structural mutations:

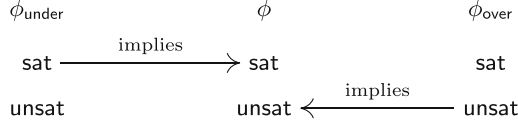
- *Small formulas are easier to solve.* In general, smaller cases have simpler structures and are presumably easier to reason about. A similar observation exists in compiler testing, where developers strongly encourage submitting small, reproducible test programs because it is easier to manually inspect small test cases [36]. Indeed, well-known production compilers such as GCC and LLVM always advocate test reduction [31] in bug reporting processes [17, 24]. Following the same observation, our structural mutations produce smaller formulas that are generally “simpler” to solve.
- *Approximations enable indirect reasoning of formulas.* Approximations, which are used in many SMT solving techniques [8, 11, 20], enables indirect reasoning of formulas. By mutating the original formula, our technique can either over- or under-approximate the original `unknown` formula. For example, we can perform a structural mutation by deleting a top-level conjunct, which relaxes the original `unknown` formula and provides an over-approximation. If the over-approximated formula is `unsat`, the origin formula must be `unsat`. Figure 1 describes the rationale for retrieving `unknown` formulas via over- and under-approximations.

## 2 Motivating Examples

This section gives two motivating examples of reasoning about `unknown` formulas via structural mutations.

Figure 2 gives a formula [2] in the LIA (Linear Integer Arithmetic) category of the SMT-LIB benchmarks. Z3<sup>1</sup> reports `unknown` on the original formula in Fig. 2

<sup>1</sup> We use commit 11477f1 (December 16, 2020) for Z3.



**Fig. 1.** Satisfiability relations between the original formula  $\phi$ , the under-approximated version  $\phi_{\text{under}}$ , and the over-approximated version  $\phi_{\text{over}}$ . Arrows in the figure represent implication relations.

due to incomplete quantifiers. We mutate the original formula by deleting the third assertion (lines 15–20, inclusive). With fewer assertions, we clearly have obtained a “relaxed” version of the original formula. Z3 can successfully report **unsat** on our over-approximated formula. Therefore, we can conclude that the original formula is unsatisfiable because even the over-approximated formula is unsatisfiable (the “ $\leftarrow$ ” direction in Fig. 1).

Figure 3 gives a formula [1] in the AUFLIA (Arrays, Uninterpreted Functions, and Linear Integer Arithmetic) category of the popular SMT-LIB benchmarks. CVC4<sup>2</sup> reports **unknown** on the original formula in Fig. 3a because the solver is incomplete in this case. We mutate the original formula by instantiating the free variable  $\mathbf{n}$  (line 3) to the constant 0. Clearly, this is an under-approximation because it restricts the value of  $\mathbf{n}$ . CVC4 can successfully solve the under-approximated formula and return **sat**. Because the under-approximated version is satisfiable, it implies that the original formula is satisfiable (the “ $\rightarrow$ ” direction in Fig. 1). Moreover, CVC4 can generate a model in Fig. 3b for the under-approximated formula. The model assigns **false** and 0 to the uninterpreted functions  $\mathbf{f}$  and  $\mathbf{v}$ , respectively. It is straightforward that appending  $\mathbf{n} = 0$  to the model gives us a model of the original formula, because if we evaluate the formula on this model, the assertion becomes “there does not exist an  $\mathbf{x}$  such that  $1 \leq x \leq 0$  and ...”, which is clearly true.

### 3 Structural Mutations

SMT formulas are first-order logic formulas with respect to different background theories. A theory over a signature  $\Sigma$  could be defined as a set  $I$  of interpretations for  $\Sigma$ , and  $I$  is also called the models of  $T$ . Under a background theory  $T$ , we use  $\phi(\vec{x})$  to represent a SMT formula with free variables  $\vec{x}$  as a vector.  $\phi(\vec{x})$  is satisfiable if and only if there exists a model of  $T$  in which  $\phi(\vec{x})$  evaluates to **true**. Otherwise, the formula is unsatisfiable. In practice, a model  $M$  of  $\phi(\vec{x})$  usually refers to a function that maps each free variable in  $\vec{x}$  to a value of the corresponding sort, such that  $\phi(\vec{x})$  evaluates to **true** under this assignment and the corresponding theory. We adopt this function-mapping view of models in later sections. Moreover, we assume a fixed background theory  $T$  over a signature  $\Sigma$ . Let  $F_{\Sigma}$  be the set of formulas over  $\Sigma$ .

<sup>2</sup> We use commit 80e0246 (December 16, 2020) for CVC4.



```

1 (set-logic LIA)
2 (declare-fun ~a29~0 () Int)
3 (assert (not (exists ((v_prenex_81 Int))
4   (let ((.cse0 (* 4 (div v_prenex_81 5))))
5     (and (<= 0 (+ .cse0 4)) (<= 0 .cse0)
6       (<= ~a29~0 (+ (mod .cse0 299978) 300021))
7       (= 0 (mod v_prenex_81 5)))))))
8 (assert (not (exists ((v~a29~0_1039 Int))
9   (let ((.cse1 (* 4 (div v~a29~0_1039 5))))
10    (let ((.cse0 (mod .cse1 299978)))
11      (and (= 0 (mod v~a29~0_1039 5))
12        (not (= 0 .cse0)) (< .cse1 0)
13        (<= ~a29~0 (+ .cse0 43))
14        (= (mod (+ .cse1 4) 299978) 0)))))))
15 (assert (not (exists ((v_prenex_81 Int))
16   (let ((.cse2 (* 4 (div v_prenex_81 5))))
17     (let ((.cse1 (+ .cse2 4)) (.cse0 (mod .cse2 299978)))
18       (and (<= ~a29~0 (+ .cse0 300021))
19         (< .cse1 0) (not (= (mod .cse1 299978) 0))
20         (= 0 .cse0) (= 0 (mod v_prenex_81 5))))))))))
21 (assert (exists ((v_prenex_81 Int))
22   (let ((.cse0 (* 4 (div v_prenex_81 5))))
23     (and (<= 0 (+ .cse0 4)) (<= 0 v_prenex_81) (<= 0 .cse0)
24       (<= ~a29~0 (+ (mod .cse0 299978) 300021))))))
25 (check-sat)
26 (exit)

```

**Fig. 2.** An over-approximation example for Z3, where the over-approximation is realized by removing the third assertion (line 15–20, inclusive).

<pre> 1 (set-logic AUFLIA) 2 (declare-fun f (Int Int) Bool) 3 (declare-fun n () Int) 4 (declare-fun v () Int) 5 (assert (! (not (exists ((x Int)) 6   (and (&lt;= 1 x) (&lt;= x n) (f x v)))) 7   :named goal)) 8 (check-sat) 9 (exit) </pre>	<pre> (   (define-fun f     ((BOUND_VARIABLE_327 Int)      (BOUND_VARIABLE_328 Int))     Bool false)   (define-fun v () Int 0) ) </pre>
---	---

(a) Original formula.

(b) A model for our under-approximated formula.

**Fig. 3.** An under-approximation example for CVC4, where the under-approximation is realized by instantiating the free variable  $n$  (line 3) to 0.

**Definition 1 (Structural Mutations).** A structural mutation  $\mathcal{M}$  is a function from  $F_\Sigma$  to  $F_\Sigma$  such that for each  $\phi \in F_\Sigma$ ,  $\mathcal{M}(\phi)$  could be obtained by replacing  $n$  ( $n > 0$  and  $n$  may depend on  $\phi$ ) non-overlapping subterms  $f_1, f_2, \dots, f_n$  in  $\phi$  with  $n$  new terms  $g_1, g_2, \dots, g_n$  simultaneously, where for each  $i \in \{1, 2, \dots, n\}$ ,  $f_i$  and  $g_i$  are of the same sort.

The essence of structural mutations is approximating **unknown** formulas. The usefulness of retrieved satisfiability results is strongly correlated to the approximation directions. Based on Fig. 1, if solvers return **sat** for over-approximated formulas  $\phi_{\text{over}}$ , the result is uninformative. Similarly, the **unsat** result from under-approximated formulas  $\phi_{\text{under}}$  is uninformative as well. Straightforward and unguided approximations can easily lead to uninformative results. In the ideal case, approximations achieved by structural mutations need to be *effec-*

*tive* (i.e., they could make **unknown** formulas solvable) and *admissible* (i.e., they should not lead to uninformative results).

Unfortunately, there is a tension between effectiveness and admissibility, and finding a sweet spot of approximations is challenging. To tackle the challenge, we devise *fine-grained* mutations to strike a balance between these two competing needs. Specifically, by repeatedly applying small structural mutations to the original formula, we get a directed acyclic graph (DAG) of mutated formulas whose nodes are formulas and edges are approximation steps. The graph is acyclic because our mutations strictly reduce formulas. Then, based on the satisfiability results of running solvers on mutated formulas, we can perform a backtracking search on this DAG to refine the approximated formulas  $\phi'$ . Consequently, the feedback-based iteration guides structural mutations toward the useful directions (depicted as “ $\xrightarrow{\text{“implies”}}$ ” and “ $\xleftarrow{\text{“implies”}}$ ”) in Fig. 1. Our fine-grained mutation process resembles abstraction refinements. However, common abstraction refinement techniques for SMT solvers (e.g. the mixed abstraction technique [8]) are not directly applicable because they (1) do not explicitly handle the **unknown** cases and (2) finally, always resort to the most precise abstraction (the original formula) but in our case, the original formula is **unknown**.

We propose four concrete structural mutations. The mutations are both reducers and approximations (i.e., they can both reduce and approximate the original formulas). Moreover, they are all theory-independent, meaning that they could be applied to all background theories. In our mutations, a top-level disjunct/conjunct denotes a disjunct/conjunct whose corresponding disjunction/conjunction is at the root of the formula’s abstract syntax tree (AST). For example, in  $P \vee Q$ ,  $P$  is a top-level disjunct. A non-trivial disjunct/conjunct is a disjunct/conjunct that is not the literal **false/true**. A non-trivial subterm is a subterm that is not a single free variable.

- *Removing Top-Level Disjuncts* ( $\mathcal{U}_\vee$ ): Replacing the first top-level non-trivial disjunct (if it exists) with **false** is a structural mutation. It is a reducer with respect to the number of top-level non-trivial disjuncts. It is also a domain-preserving under-approximation. Note that changing  $P \vee Q$  to **false**  $\vee Q$  could still be regarded as domain-preserving, because **false**  $\vee Q$  could be regarded as a formula with free variables  $\{P, Q\}$  while  $P$  is not used.
- *Instantiating Free Variables* ( $\mathcal{U}_\text{in}$ ): Replacing all occurrences of the first occurred free variable (if it exists) with one value in its sort is a structural mutation. It is a reducer with respect to the number of free variables. It is also a domain-adjusting under-approximation.
- *Removing Top-Level Conjuncts* ( $\mathcal{O}_\wedge$ ): Replacing the first top-level non-trivial conjunct (if it exists) with **true** is a structural mutation. It is a reducer with respect to the number of top-level non-trivial conjuncts. It is also a domain-preserving over-approximation.
- *Abstracting Subterms* ( $\mathcal{O}_\text{term}$ ): Replacing the first non-trivial subterm that does not contain variables bound by quantifiers (if it exists) with a new free variable of the same sort is a structural mutation. It is a reducer with respect

to the number of non-trivial subterms. It is also a domain-adjusting over-approximation.

Note that some mutation steps could be interpreted as several different approximations. For example, replacing  $P$  in  $P \wedge Q$  with `true` could be regarded as a domain-adjusting under-approximation  $\mathcal{U}_{\text{in}}$  or a domain-preserving over-approximation  $\mathcal{O}_{\wedge}$ . The actual effect is preserving the satisfiability because both the original formula  $P \wedge Q$  and the modified formula `true`  $\wedge Q$  are satisfiable.

The definitions in Sect. 3 impose constraints such as “replacing the first top-level disjunct” when there are multiple top-level disjuncts, so each application of mutation produces only one transformed formula. It is possible to remove those constraints and get multiple mutated formulas in each step. Therefore, we can get more mutated formulas to use in practice. If we regard the mutated formulas as nodes and approximation steps as directed edges, we form a directed acyclic graph (DAG). It is a DAG because there is no cycle due to the reducer property. We call the graph “under-approximation DAG” or “over-approximation DAG”.

Our unknown formula retrieval algorithms repeatedly apply and revert mutations on the original formula. Thus, it forms a process of *adjusting* the approximations (making more or fewer approximations) along the corresponding under- or over-approximation DAG. Recall that approximations can be uninformative (e.g. over-approximating a formula  $\phi$  to a `sat` formula  $\phi'$  is uninformative since it provides no information about  $\phi$ ). To avoid encountering too many uninformative cases, our structural mutation framework prunes the adjusting process based on over- or under-approximation DAG: if we over-approximate the formula  $\phi$  to a satisfiable formula, we don’t need to continue the current branch of over-approximation since further over-approximations can only produce uninformative approximations. Similarly, if we under-approximate the formula  $\phi$  to an unsatisfiable formula, we can stop the current branch of under-approximation.

## 4 Related Work

Formula simplification techniques have been developed to simplify formulas for SMT solvers [14, 33, 34]. These techniques, however, produce equivalent or equisatisfiable formulas, and thus often need to do sophisticated reasoning about Boolean logic and underlying theories. Our structural mutations relax the requirement from equivalence or equisatisfiability to approximations, and thus produce transformations that are easier to reason about.

Approximations have also been widely used to solve SMT formulas. The DPLL(T) framework [15, 28, 29], which forms the basis of many modern SMT solvers, leverages the Boolean abstraction of the original formula and then refines the abstraction using information provided by theory-specific solvers. De Moura and Rueß [27] have proposed lemmas on demand, which is also an abstraction refinement process. Approximations can also be done in the theory/first-order layer [10, 26]. Bauer et al. have proposed a technique that can ignore parts of the Boolean abstraction that do not affect the overall truth value [6]. Explicit

approximations have been introduced to SMT solvers to model bit-vector operations [20, 37] and bit-vector values [9]. SMT solvers can also alternate between over-approximations and under-approximations [11, 21, 22], as well as mixing them altogether [8]. Approximations also help to simplify formulas [33], to change the decidability of certain formulas [16], etc. In the refinement aspect, techniques similar to counter-example guided abstraction refinement [13] are well-developed in SMT solvers. Approximating formulas can also happen outside solvers. For example, concolic testing [18, 35] simplifies formulas by instantiating variables before using solvers to solve them. Compared with those existing techniques, our structural mutations are solver/theory-independent, are not part of any solver or automated reasoning tools, and can be applied to almost all types of formulas.

## 5 Conclusion

This paper has discussed a source-level approach to improve SMT solving: instead of improving solvers for all possible input formulas, we focus on mutating (approximating) formulas that are already **unknown** to solvers. As the next step, we plan to conduct an extensive study to validate the idea on real-world SMT constraints.

**Acknowledgements.** We thank the anonymous reviewers for their feedback. This work was supported, in part, by the United States National Science Foundation (NSF) under grants No. 2114627 and No. 2237440; and by the Defense Advanced Research Projects Agency (DARPA) under grant N66001-21-C-4024. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsoring entities.

## References

1. A Test Case of AUFLIA (Arrays, Uninterpreted Functions, and Linear Integer Arithmetic) Logic. <https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/AUFLIA/-/blob/master/20170829-Rodin/smt4391808662368180273.smt2>. Accessed Jan 2021
2. A Test Case of LIA (Linear Integer Arithmetic) Logic. [https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/LIA/-/blob/master/20190429-UltimateAutomizerSvcomp2019/Problem15\\_label00\\_false-unreach-call.c\\_5.smt2](https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/LIA/-/blob/master/20190429-UltimateAutomizerSvcomp2019/Problem15_label00_false-unreach-call.c_5.smt2). Accessed Jan 2021
3. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). [https://doi.org/10.1007/978-3-030-99524-9\\_24](https://doi.org/10.1007/978-3-030-99524-9_24)
4. Barbosa, H., Hoenicke, J., Hyvarinen, A.: 15th International Satisfiability Modulo Theories Competition (SMT-COMP 2020): Rules and Procedures. <https://smt-comp.github.io/2020/rules20.pdf>. Accessed Feb 2021

5. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-22110-1\\_14](https://doi.org/10.1007/978-3-642-22110-1_14)
6. Bauer, A., Leucker, M., Schallhart, C., Tautschnig, M.: Don't care in SMT: building flexible yet efficient abstraction/refinement solvers. *Int. J. Softw. Tools Technol. Transf.* **12**(1), 23–37 (2010)
7. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. *J. Autom. Reason.* **60**(3), 299–335 (2018)
8. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2009)*, pp. 69–76 (2009)
9. Brummayer, R., Biere, A.: Effective bit-width and under-approximation. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) EUROCAST 2009. LNCS, vol. 5717, pp. 304–311. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-04772-5\\_40](https://doi.org/10.1007/978-3-642-04772-5_40)
10. Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. *J. Satisf. Boolean Model. Comput.* **6**(1–3), 165–201 (2009)
11. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding bit-vector arithmetic with abstraction. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 358–372. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-71209-1\\_28](https://doi.org/10.1007/978-3-540-71209-1_28)
12. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, pp. 209–224 (2008)
13. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, pp. 154–169 (2000)
14. Dillig, I., Dillig, T., Aiken, A.: Small formulas for large programs: on-line constraint simplification in scalable static analysis. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 236–252. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-15769-1\\_15](https://doi.org/10.1007/978-3-642-15769-1_15)
15. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL( T): fast decision procedures. In: *Proceedings of the 16th International Conference Computer Aided Verification (CAV 2004)*, pp. 175–188 (2004)
16. Gao, S., Avigad, J., Clarke, E.M.: Delta-decidability over the reals. In: *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science (LICS 2012)* pp. 305–314 (2012)
17. GCC: A Guide to Testcase Reduction. [https://gcc.gnu.org/wiki/A\\_guide\\_to\\_testcase\\_reduction](https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction). Accessed Jan 2021
18. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, IL, USA, June 12–15, 2005, pp. 213–223. ACM (2005)
19. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)*, pp. 215–224 (2010)
20. Jonáš, M., Strejček, J.: Abstraction of bit-vector operations for BDD-based SMT solvers. In: Fischer, B., Uustalu, T. (eds.) ICTAC 2018. LNCS, vol. 11187, pp. 273–291. Springer, Cham (2018). [https://doi.org/10.1007/978-3-030-02508-3\\_15](https://doi.org/10.1007/978-3-030-02508-3_15)

21. Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O.: Abstraction-based satisfiability solving of presburger arithmetic. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 308–320. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-27813-9\\_24](https://doi.org/10.1007/978-3-540-27813-9_24)
22. Lahiri, S.K., Mehra, K.K.: Interpolant based decision procedure for quantifier-free presburger arithmetic. *J. Satisf. Boolean Model. Comput.* **1**(3–4), 187–207 (2007)
23. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16), pp. 348–370 (2010)
24. LLVM: How to submit an LLVM bug report. <https://llvm.org/docs/HowToSubmitABug.html>. Accessed Jan 2021
25. de Moura, L., Björner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24)
26. de Moura, L., Jovanović, D.: A model-constructing satisfiability calculus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 1–12. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-35873-9\\_1](https://doi.org/10.1007/978-3-642-35873-9_1)
27. de Moura, L., Rueß, H.: Lemmas on demand for satisfiability solvers. In: Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2002), pp. 244–251 (2002)
28. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with exhaustive theory propagation and its application to difference logic. In: Etesami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005). [https://doi.org/10.1007/11513988\\_33](https://doi.org/10.1007/11513988_33)
29. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract davis-putnam-logemann-loveland procedure to DPLL(T). *J. ACM* **53**(6), 937–977 (2006)
30. Palikareva, H., Cadar, C.: Multi-solver support in symbolic execution. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 53–68. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-39799-8\\_3](https://doi.org/10.1007/978-3-642-39799-8_3)
31. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012), pp. 335–346 (2012)
32. Reynolds, A., et al.: Rewrites for SMT Solvers Using Syntax-Guided Enumeration. <http://homepage.divms.uiowa.edu/~ajreynol/pres-smt2018.pdf>. Accessed Feb 2021
33. Reynolds, A., Nötzli, A., Barrett, C., Tinelli, C.: High-level abstractions for simplifying extended string constraints in SMT. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 23–42. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-25543-5\\_2](https://doi.org/10.1007/978-3-030-25543-5_2)
34. Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 453–474. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-63390-9\\_24](https://doi.org/10.1007/978-3-319-63390-9_24)
35. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005, pp. 263–272. ACM (2005)

36. Sun, C., Li, Y., Zhang, Q., Gu, T., Su, Z.: Perses: syntax-guided program reduction. In: Proceedings of the 40th International Conference on Software Engineering (ICSE 2018), pp. 361–371 (2018)
37. Teuber, S., Büning, M.K., Sinz, C.: An incremental abstraction scheme for solving hard smt-instances over bit-vectors. CoRR **abs/2008.10061** (2020). <https://arxiv.org/abs/2008.10061>



# On the Cloud We Can't Wait: Asynchronous Actors Perform Even Better on the Cloud

Aniruddha Mysore<sup>(✉)</sup>, Youssef Elmougy, and Akihiro Hayashi

Georgia Institute of Technology, Atlanta, USA  
{animysore,yelmougy3,ahayashi}@gatech.edu

**Abstract.** This study investigates the performance of asynchronous actor programming and synchronous Partitioned Global Address Space (PGAS) versions of graph kernels on a Cloud platform and a High-Performance Computing (HPC) platform. Using the Bale suite of graph microkernels, we compare the execution times of kernels implemented with OpenSHMEM (synchronous PGAS) and HClib-actor (asynchronous) on both Azure Cloud with Ethernet and an HPC cluster with InfiniBand. Our results reveal significant performance differences between these platforms. While the asynchronous version outperforms the synchronous version in both settings, the performance gap is dramatically wider on the Cloud platform, with the asynchronous version showing up to 1,000x improvement over the synchronous version in some cases. Moreover, we observe highly variable execution times in the Cloud, likely due to shared resource interference and unpredictable data center traffic. These findings highlight the importance of choosing appropriate programming models for different computational platforms, especially as Cloud platforms are becoming more affordable and easier to access compared to traditional HPC clusters. Our work provides valuable insights for both researchers and practitioners in optimizing parallel programming strategies across diverse computational settings.

**Keywords:** The Actor Model · PGAS · Cloud Computing · HPC · High-Performance Graph Analytics

## 1 Introduction

### 1.1 Background

Large-scale clusters have grown rapidly in recent times - a phenomenon understood to be the consequence of the end of Moore's law and Dennard scaling, which has led to limits on the potential performance gain from a single chip. This has led to the rise of scalable programming models that can utilize the capabilities of such clusters effectively. In particular, there has been a growing interest in employing the Partitioned Global Address Space (PGAS) model [13], which gives the programmer an illusion of shared memory programming for such large-scale platforms.



Coordination between nodes in PGAS is typically done with Bulk-Synchronous-Parallel (BSP) communication, but recent work [11,12] shows the promise of enabling actor-based fine-grained asynchronous messaging with message aggregation in large-scale graph applications. Specifically, [11,12] show that the abstraction of asynchronous actor-selector programming allows graph algorithms to be expressed very efficiently while allowing for intuitive programming. HClib-actor [11] is a PGAS runtime that leverages the concept of selectors - independent computational entities, or actors, that communicate via message passing - to achieve high levels of concurrency without the pitfalls of traditional locking mechanisms. Selectors are derived from [8] and are a modification of the more conventional actor model (originally proposed in the 1970s [7]), where each actor possesses multiple mailboxes.

Cloud computing and high-performance computing (HPC) clusters represent two prevalent environments where these programming strategies are deployed. Cloud computing offers scalable, on-demand resources via platforms such as Azure, AWS, GCP, and numerous others, coupling flexible resource allocation and cost efficiency. Historically, Cloud infrastructures were predominantly utilized for internet services, which relied on inexpensive off-the-shelf hardware, with their computing capabilities not matching those of dedicated HPC clusters. This disparity was not only due to the differences in processing power but also due to the faster communication networks used in clusters, like Infiniband, compared to the slower Ethernet-based connections in Cloud setups. However, recent advancements have significantly bridged this gap. Improvements in Cloud technologies, such as the availability of specialized interconnects like Mellanox Infiniband in the public Cloud, have enhanced communication efficiencies, allowing Cloud infrastructures to become more comparable to traditional clusters. Despite these advancements, the Cloud environment still poses unique challenges, such as issues related to virtualization and the non-deterministic nature of locality, which can significantly affect performance. Conversely to the heterogeneity of the Cloud, HPC clusters are characterized by dedicated, largely homogeneous hardware and optimized network topologies designed for maximum performance and efficiency in executing large-scale computations.

## 1.2 Motivation

The HPC community has identified that network noise in the Cloud can be a performance bottleneck in previous studies [6]. In this study, we build upon past works by seeking to understand *how composing HPC programs, either using asynchronous actor programming or synchronous SHMEM models, affects their performance on the Cloud versus HPC clusters*. Industry practitioners and researchers often assume that the performance characteristics observed in one environment will translate to another, yet this is not always the case. The unique architectural and operational differences between Cloud platforms and HPC clusters can significantly impact the performance of parallel programming models.

Our preliminary experiments have shown that the performance difference between asynchronous message passing and blocking strategies is more pro-

nounced in the Cloud compared to traditional HPC clusters. This observation prompts a deeper investigation into the factors contributing to this discrepancy. Understanding these factors is crucial for making informed decisions about deploying concurrent systems in various environments and optimizing performance and resource utilization.

### 1.3 Objectives

This work studies the performance of asynchronous actor programming and synchronous PGAS (SHMEM) strategies in Cloud environments versus HPC clusters. More specifically, we:

- measure and compare the performance of graph kernels written in HCLib-actor (an asynchronous actor programming runtime) and an equivalent non-asynchronous version on both Azure Cloud and HPC clusters.
- identify and analyze the factors contributing to the observed performance differences in these environments and provide insights and recommendations for industry practitioners on deploying these programming strategies effectively in different computational settings.

By addressing these objectives, this paper seeks to bridge the knowledge gap between theory and practice in the realm of HPC-on-the-Cloud, offering valuable guidance to academia and industry. The findings of this study are expected to inform future research and development in optimizing the deployment of asynchronous and blocking strategies in diverse computational environments.

## 2 Related Work

### 2.1 Partitioned Global Address Space (PGAS)

PGAS [1, 13] is a parallel programming model that provides a global memory address space partitioned among the processors. Languages such as Unified Parallel C (UPC) [3], Chapel [2], and X10 [5] have implemented PGAS concepts to simplify the memory access semantics, thereby enhancing the productivity and performance of parallel application development.

The OpenSHMEM programming model [4], one implementation of PGAS, provides efficient one-sided communication primitives that allow a process to directly access the memory of another process without the involvement of the target process’s CPU. This feature is particularly beneficial for achieving low-latency communication in HPC environments.

### 2.2 Asynchronous Actor Programming

The asynchronous Actor programming model has been the subject of active research and development for several decades. Initially introduced in the late

1970s, it is designed to provide a natural abstraction for concurrent computation. In this model, “actors” are the fundamental units of computation that encapsulate state and behavior, communicate via asynchronous message passing, and make decisions based on the messages they receive. This decoupling of computation and communication helps achieve high concurrency and scalability levels.

Recent advancements in Actor-based programming frameworks, such as Akka, Erlang, and Orleans, have demonstrated the effectiveness of this model in building robust, scalable, and fault-tolerant distributed systems. These frameworks leverage the actor model to manage complex concurrency issues, making them popular choices for developing distributed applications in industry and research settings. The actor model’s adaptability to various domains, including real-time data processing, telecommunications, and Cloud computing, underscores its relevance and potential.

### 2.3 Cloud Environments

With the growing adoption of Cloud computing for large-scale computational tasks, there has been increasing interest in evaluating the performance of parallel programming models in Cloud environments. Cloud platforms such as Microsoft Azure, Amazon Web Services (AWS), and Google Cloud offer scalable and cost-effective solutions for running parallel applications. Though, they still introduce new challenges, such as network virtualization, variable latency, and resource contention.

Little prior research explores how these Cloud-specific factors impact the performance of asynchronous actors and PGAS models. The authors believe that Cloud environments’ inherent elasticity and resource abstraction can lead to performance differences from those observed in traditional HPC settings. For example, the virtualization layer in Cloud platforms can introduce additional latency, which may affect the efficiency of synchronous communication patterns more than asynchronous ones.

### 2.4 Gaps and Opportunities

While there is extensive literature on the performance of asynchronous actor and PGAS programming models in HPC environments, relatively few studies focus on their performance in Cloud computing contexts.

This study aims to fill this gap by providing a detailed comparative analysis of asynchronous actor-based programming (HCLib-actor) and PGAS programming (OpenSHMEM) on the Azure Cloud platform. By focusing on graph microkernels, which are representative of a wide range of real-world applications, this research offers actionable insights for industry practitioners looking to optimize Cloud-based applications and academics interested in advancing the state of parallel computing research.

### 3 Methodology

#### 3.1 Experimental Setup

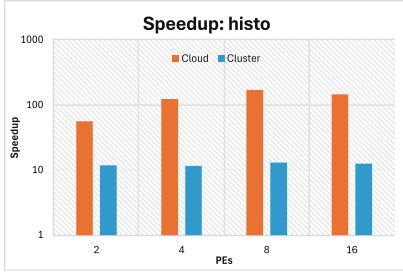
Our experiments involve running the same set of kernels on a Cloud platform and an HPC cluster. We use the PACE [10] cluster for the HPC platform and Azure D-series virtual machines for the Cloud platform. Since we are focused on the latency overheads introduced by the network, we utilize only one core per machine on both platforms. PACE nodes use Intel Xeon Gold 6226 CPUs, while Cloud nodes utilize Intel Xeon Platinum 8473C CPUs. PACE is equipped with InfiniBand networking; all nodes are in the same data center. The networking between Cloud nodes is far more heterogeneous and more liable to changing data center traffic. There exists only a region-level locality guarantee. Finally, to keep findings general to most Cloud networks, we chose not to enable Azure-only networking features like single-root input/output virtualization (SR-IOV) or proximity placement. The latter does not offer locality guarantees beyond best effort and, in practice, the performance benefit observed was small enough that we chose to turn it off to make our results more generalizable. In the same spirit of targeting a “generic cloud” machine and network, we decided not to use Azure’s “HPC-optimized” offering, which includes a setup very similar to an HPC Cluster, such as having Infiniband networking between nodes. The D-series VMs we picked for experiments use Ethernet.

#### 3.2 Benchmark and Metric

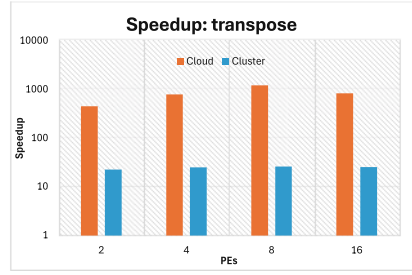
In this study, we employ the Bale suite of graph microkernels to evaluate the performance of asynchronous actor-based programming and blocking PGAS models. The Bale suite includes seven kernels designed to capture a wide range of irregular access patterns common in many applications:

1. Histogram: Builds histogram from a set of randomly generated data, some remote and some local.
2. Index Gather: Gathers elements from a (remote) array based on a list of indices.
3. Permute: Permutes the elements of an array.
4. Randperm: Generates and stores a random permutation of array elements.
5. Transpose: Transposes a large, sparse matrix.
6. Triangle: Count the number of triangles (three-degree cycles) in a sparse graph.
7. Toposort: Topological sort on a sparse DAG.

These benchmarks collectively cover a variety of computational tasks over sparse matrices. For each kernel, we study three PGAS versions - one that is implemented in OpenSHMEM (and uses synchronous communication) and two asynchronous versions - one implemented using the Conveyors message aggregation library [9], while the other uses the HCLib-actor runtime [11, 12] which is built on top of the Conveyors library. The measured metric is execution time in seconds.



(a) Speedup on histogram kernel



(b) Speedup on matrix transpose kernel

**Fig. 1.** Comparing relative Speedup of Selector over OpenSHMEM variant for two graph kernels on an HPC Cluster and the Cloud. Note that the vertical axis is logarithmic.

## 4 Experimental Observations

This section details our observations from running the Bale suite on both platforms. To restate our goal, we are empirically studying the question: **How many times faster is the selector version compared to the synchronous OpenSHMEM version?** The speedup comparisons across these two platforms are depicted in charts for two kernels in Fig. 1. Aggregates of execution time for each kernel across three programming models and both platforms are provided in Table 2. Charts depicting this data are provided in Fig. 2 - since the difference in execution times between kernel versions is extreme on the Cloud, we have used a logarithmic axis (base-10) for the columns on the right, except for the very first figure where we use a linear scale to show the standard error better.

**Table 1.** Speedup gained from asynchronous actors over Bale graph kernels. Speedup on the HPC cluster is displayed in the shaded (blue) columns and Speedup on the Cloud is displayed in unshaded (white) columns. Speedup is computed as the quotient resulting from the selector version running time divided by the OpenSHMEM version running time from Table 2.

Kernel	Number of PEs							
	2	2	4	4	8	8	16	16
histogram	11.9	56.5	11.6	124.2	13.1	171.7	12.6	146.2
index-gather	14	196.7	20.2	387.3	23.1	442.1	23.9	379.1
permute	13.8	249.8	20.1	582.5	22.9	704.6	23	690.9
randperm	9.8	155.8	11.3	335.7	11.7	397.9	11.5	387.6
topological sort	39	555.6	46.8	905.9	50.5	921.6	48.2	380.9
matrix transpose	22.7	451.2	25	792.7	26.1	1206.9	25.5	833.9
triangle counting	15.7	356	22.6	725.3	26	914.5	27.1	876.5

Table 1 provides speedup data across the Cloud and Cluster platforms for all kernels.

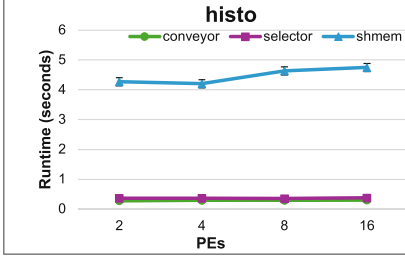
From the observations, the easy-to-predict insight is that the HPC cluster is faster than the Cloud due to the cluster having a dedicated interconnect. However, this study also revealed several interesting performance facts, including the variation in execution times on the Cloud and the incredible improvements wrested by switching to asynchronous communication on such slower networks.

**Table 2.** Bale suite running times on an HPC cluster and a Cloud environment. All running times are in seconds. Cluster running times are in shaded (blue) columns and Cloud running times are in unshaded (white) columns. All values are computed as averages of five executions.

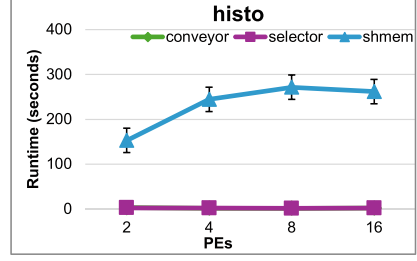
Average running time per kernel		Number of PEs							
Kernel	Version	2	2	4	4	8	8	16	16
histogram	conveyor	0.2832	2.8876	0.2926	1.7036	0.2934	2.023	0.3032	6.3998
	selector	0.3608	2.7106	0.3644	1.9702	0.3548	1.7934	0.3798	5.0764
	shmem	4.2676	153.053	4.202	244.629	4.6294	262.0524	4.7478	608.085
index-gather	conveyor	0.6378	9.0226	0.6548	6.9162	0.662	6.0698	0.6724	6.7602
	selector	0.8116	8.633	0.815	6.7688	0.82	6.2924	0.85	7.5202
	shmem	11.3134	1697.404	16.3852	2620.942	18.9024	2781.248	20.2506	2850.5
permute	conveyor	0.285	2.4858	0.2922	2.5552	0.2892	1.9004	0.3132	2.3514
	selector	0.3304	2.6586	0.3312	1.8768	0.3376	1.695	0.3606	2.029
	shmem	4.5272	664.108	6.6486	1093.146	7.6992	1194.291	8.2578	1401.715
randperm	conveyor	0.2338	1.3418	0.2396	1.0888	0.2416	1.1378	0.2558	1.4236
	selector	0.265	1.6654	0.2682	1.2648	0.273	1.3332	0.2872	1.51
	shmem	2.594	259.461	3.007	424.518	3.1876	530.468	3.2786	585.274
topological sort	conveyor	0.1088	1.2472	0.1112	1.189	0.1128	1.344	0.1162	2.3386
	selector	0.1464	1.2074	0.1492	1.1246	0.1518	1.2526	0.1676	3.172
	shmem	5.7026	670.719	6.979	1018.718	7.6578	1154.388	8.0726	1207.932
matrix transpose	conveyor	0.2568	1.4614	0.2584	1.0742	0.2606	1.1566	0.2758	1.6644
	selector	0.2824	1.422	0.2874	1.2322	0.2918	0.934	0.3072	1.578
	shmem	6.3982	641.528	7.1758	976.745	7.588	1127.212	7.8224	1315.747
triangle counting	conveyor	0.5196	3.8704	0.5206	3.3188	0.5176	2.8854	0.5398	2.9268
	selector	0.6004	4.1872	0.604	3.2088	0.6088	2.59075	0.6266	3.0792
	shmem	9.3682	1490.288	13.6376	2327.121	15.7818	2369.132	16.955	2698.743

**Broad Advantages of Networking in Sparse Graph Operations.** As can be seen in Table 2, in our experiments, all kernels perform much better on the HPC cluster, which is to be expected by virtue of the latter having an Infiniband 100HDR interconnect.

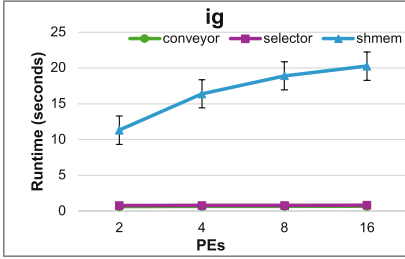
**Highly Variable Cloud Execution Times.** We observe that the workloads on the Cloud have extreme variation, especially when executing the slower kernels. For instance, the OpenSHMEM histogram kernel had a 68-second difference between the slowest and fastest observation on the Cloud. However, on



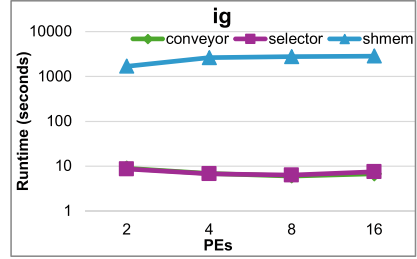
(a) Histogram on HPC Cluster.



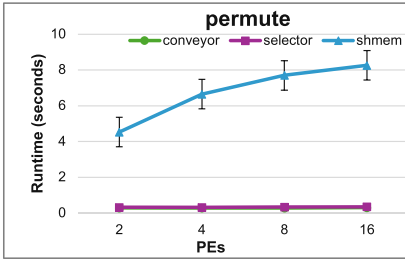
(b) Histogram on Cloud.



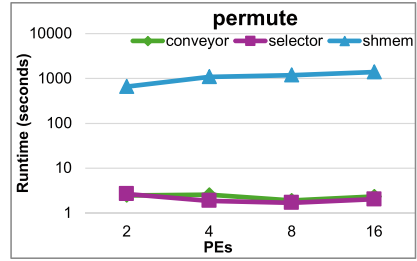
(c) Index-gather on HPC Cluster.



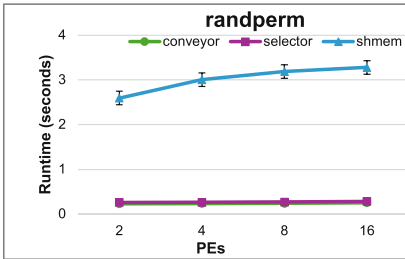
(d) Index-gather on Cloud.



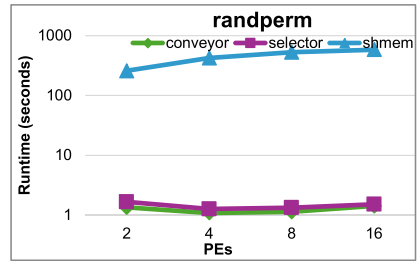
(e) Matrix permute on HPC Cluster.



(f) Matrix permute on Cloud.

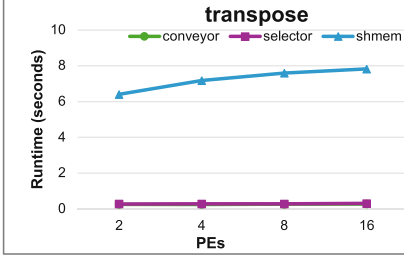


(g) Random Permutation on HPC Cluster.

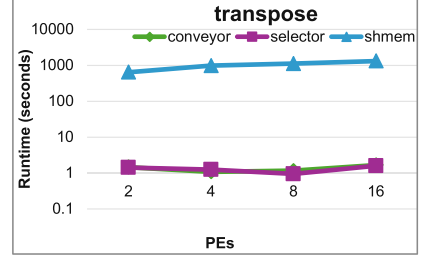


(h) Random Permutation on Cloud.

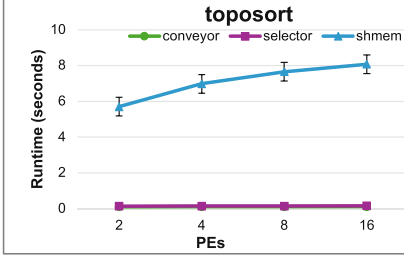
**Fig. 2.** Bale performance on the two test environments - HPC cluster (left column) and Cloud (right column). Note the vertical axes (i.e., execution times) use a linear scale in the figures in the left column and a logarithmic scale in the figures (excl. the first row) in the right column. Continued to next page.



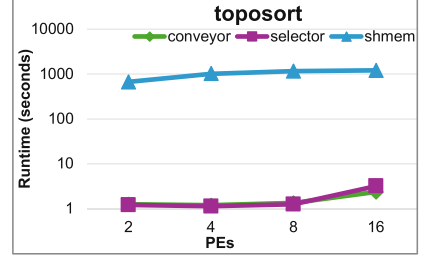
(i) Matrix Transpose on HPC Cluster.



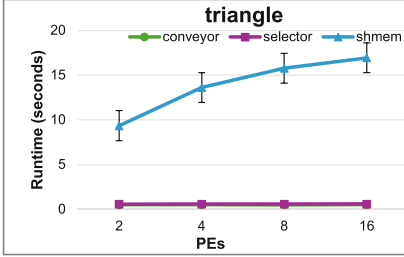
(j) Matrix Transpose on Cloud.



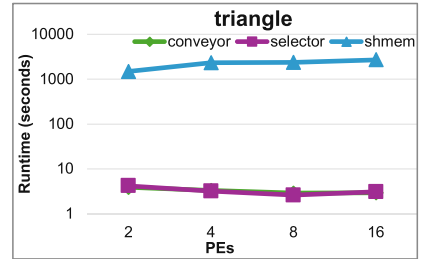
(k) Topological Sort on HPC Cluster.



(l) Topological Sort on Cloud.



(m) Triangle Counting on HPC Cluster.



(n) Triangle Counting on Cloud.

**Fig. 2.** (continued)

the Cluster, we observe that the variation between the slowest and fastest histogram kernel was in the order of milliseconds. We theorize that this is due to the non-deterministic nature of data-center traffic since we use shared resources. We control for this variation by repeating each experiment 5 times with significant delay between runs. Each value in Table 2 represents the average execution time of 5 runs while the error bars in the left columns of Fig. 2 (and in the first row of the right column) represent the variation in times.

### Orders-of-Magnitude Performance Boost with Asynchronous Actors.

The results reveal a stark benefit of using asynchronous approaches like conveyors and selectors on the Cloud; *on the HPC cluster, the selector version is at most*



*one order of magnitude ( $10 \times$  –  $30 \times$ ) faster, but this difference becomes two to three orders of magnitude ( $1000 \times$ ) when we move to the Cloud.* We attribute this performance gain to non-blocking point-to-point communication with automatic message aggregation. We highlight the speedups in Fig. 1 and Table 1.

## 5 Conclusion

Our study reveals significant performance disparities between asynchronous actor programming and synchronous PGAS strategies in Cloud versus HPC cluster environments. Asynchronous approaches show dramatically amplified benefits in Cloud settings, with performance gaps widening from one order of magnitude in HPC clusters to two or three orders in the Cloud. We also observe high execution time variability in Cloud environments, underscoring the challenges of shared resources and fluctuating data center traffic. These findings emphasize the critical role of network infrastructure in sparse graph operations and suggest that asynchronous programming models should be preferred for Cloud-based HPC applications, especially for workloads with irregular access patterns.

### 5.1 Opportunities and Future Work

This work could be extended to perform cost modeling or estimation as an additional dimension to the performance executions we have presented. Future work could augment this study with the dollar values of running these kernels on each test environment. This study also limits itself to the most generally available type of virtual machine on the Cloud, where future work could study how specific configurations (usually restricted to a particular vendor) could offer performance improvements. Lastly, future work can focus on developing adaptive runtime systems to handle the unpredictable nature and elasticity of shared Cloud resources and further optimize asynchronous programming models for Cloud environments.

**Acknowledgments.** We acknowledge Microsoft’s generous support of Azure credits made available for this research via GT Cloud Hub. The authors would also like to thank Professor Vivek Sarkar of the College of Computing, Georgia Institute of Technology, for his support of this work.

## References

1. Almasi, G.: PGAS (Partitioned Global Address Space) Languages, pp. 1539–1545. Springer US, Boston, MA (2011). [https://doi.org/10.1007/978-0-387-09766-4\\_210](https://doi.org/10.1007/978-0-387-09766-4_210)
2. Callahan, D., Chamberlain, B.L., Zima, H.P.: The cascade high productivity language. Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings, pp. 52–60 (2004). <https://api.semanticscholar.org/CorpusID:5217126>

3. Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K.A., Brooks, E.D., Warren, K.H.: Introduction to UPC and language specification (2000). <https://api.semanticscholar.org/CorpusID:59868665>
4. Chapman, B., et al.: Introducing openSHMEM: SHMEM for the PGAS community. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model. PGAS '10, Association for Computing Machinery, New York, NY, USA (2010). <https://doi.org/10.1145/2020373.2020375>
5. Charles, P., et al.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 519–538. OOPSLA '05, Association for Computing Machinery, New York, NY, USA (2005). <https://doi.org/10.1145/1094811.1094852>
6. De Sensi, D., De Matteis, T., Taranov, K., Di Girolamo, S., Rahn, T., Hoeffer, T.: Noise in the clouds: Influence of network performance variability on application scalability. *Proc. ACM Meas. Anal. Comput. Syst.* **6**(3) (Dec 2022). <https://doi.org/10.1145/3570609>
7. Hewitt, C.E., Bishop, P.B., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: International Joint Conference on Artificial Intelligence (1973). <https://api.semanticscholar.org/CorpusID:18601146>
8. Imam, S.M., Sarkar, V.: Selectors: actors with multiple guarded mailboxes. In: Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control, pp. 1–14. AGERE! '14, Association for Computing Machinery, New York, NY, USA (2014). <https://doi.org/10.1145/2687357.2687360>
9. Maley, F.M., DeVinney, J.G.: Conveyors for streaming many-to-many communication. In: 2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3), pp. 1–8. IEEE (2019)
10. PACE: Partnership for an Advanced Computing Environment (PACE) (2017). <http://www.pace.gatech.edu>
11. Paul, S.R., Hayashi, A., Chen, K., Elmougy, Y., Sarkar, V.: A fine-grained asynchronous bulk synchronous parallelism model for PGAS applications. *J. Comput. Sci.* **69**, 102014 (2023). <https://doi.org/10.1016/j.jocs.2023.102014>, <https://www.sciencedirect.com/science/article/pii/S1877750323000741>
12. Paul, S.R., Hayashi, A., Chen, K., Sarkar, V.: A productive and scalable actor-based programming system for PGAS applications **13350**, 233–247 (2022). [https://doi.org/10.1007/978-3-031-08751-6\\_17](https://doi.org/10.1007/978-3-031-08751-6_17)
13. Yelick, K., et al.: Productivity and performance using partitioned global address space languages. In: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, pp. 24–32. PASCO '07, Association for Computing Machinery, New York, NY, USA (2007). <https://doi.org/10.1145/1278177.1278183>



# A Formal Model for Portable, Heterogeneous Accelerator Programming

Zachary J. Sullivan<sup>(✉)</sup> and Samuel D. Pollard

Sandia National Laboratories, Livermore, CA, USA  
{zsulliv, spolla}@sandia.gov

**Abstract.** Programming on modern computer architectures requires logic to utilize both multi-threaded CPUs and accelerators such as GPUs. This can be fraught with errors relating to transmitting and accessing memory not available to all compute resources. Moreover, once the programmer writes correct code for one system, it is often slow or incorrect when run on a different architecture. A bottom-up approach to solving this problem is reified in the C++ library Kokkos. We approach the problem top-down, distilling and generalizing concepts found therein. We design a small language, called H-IMP—which builds on an earlier model of Kokkos called MiniKokkos—with a type system that includes notions of device memory, accelerators, and safe memory access. We show that a well-typed program is safe, which in this context means that there are no heterogeneous memory errors. Our type system enables us to define a precise notion of a *portable* program as a program with free variables representing where data is stored and kernels are executed. Finally, we prove a *portability theorem* for heterogeneous programs: that the program can run safely when instantiated on a specific set of architectures.

**Keywords:** programming languages · high-performance computing · heterogeneous computing · portability

## 1 Introduction

Modern compute nodes are structured with a host CPU together with other kinds of accelerators, typically GPUs. While writing any programs that exploit this hardware is already a difficult task, writing programs that are also meant to be portable is compounded by the wide variety of GPU and on-CPU accelerator systems. To ease programming with such machines and to abstract over the different hardware architectures, there exist many libraries and languages which offer a programming model to handle multiple parallel architectures abstractly [2–6, 8–10, 13, 15]. In Kokkos [6, 13]—a library designed specifically for portability—accelerators are abstracted into a notion of *execution spaces* that we can run kernels on; an obvious example would be a GPU, but another example is an OpenMP kernel running on the CPU. To abstract different kinds of memory accessible to different execution spaces, Kokkos has *memory spaces*;

for example, a GPU will have its on-chip memory. Importantly, these have different performance and accessibility properties depending on which execution space is being used. The host code, and only the host code, can allocate objects which exist in these memory spaces; Kokkos calls these objects *views*.

While Kokkos provides an abstraction that enables portability, using the C++ library alone does not give us the extra reasoning to know whether our code is indeed portable. Consider the following program:

```
Kokkos::View<int *, Kokkos::HostSpace> view ("V", 32);
Kokkos::parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    view(index) = index;
});
```

Implicit in this code is where the parallel for-loop is executed. Behind the scenes, this location is chosen by a configured default; this is how Kokkos code can be instantiated for different systems. If this execution space is configured to be some on-CPU space like OpenMP, then this code will run without issue. However, there exist instantiations of this default that will produce problems; for instance, using a CUDA execution space will result in a memory error when the code attempts to write to `view`. Thus, the code is only portable to a specific subset of systems. Kokkos allows us to avoid declaring the memory space explicitly and it will choose the memory space so that it matches the execution space, but then a portable program must include copying between host and this memory space.

To describe a portable, heterogeneous program as a formal property, we develop a small, formal language including these features alone. Our language includes a type system that takes from two lines of work: region-based memory management and security type systems. First, our system can be seen as a modification of the region calculus [1, 12] wherein locations are added to variables to automatically handle allocation and deallocation of objects. Our type system also adds locations, *i.e.* memory spaces, to where Kokkos views are stored. Second, we take inspiration from languages with features for information-flow security [11] wherein code is tagged with either low or high security to restrict permissions. In Kokkos, we think of code being tagged with an execution space that restricts its permission to access certain memory spaces and operations. Portability can then be defined by polymorphism over spaces in a manner which respects these permissions.

Previous work [7] on modeling Kokkos as a small programming language, called MiniKokkos addressed the problem of deadlocks. Our language H-IMP, simplifies their execution model to focus on heterogeneous memory and device permissions. Our contributions include the following:

- A core language (H-IMP) for heterogeneous hardware (Sect. 2).
- An operational semantics that captures notions of different kernel-executing machines within a global execution of a program (Sect. 3).

$$\begin{aligned}
\chi &\in \textit{Execution Space} ::= \textit{Host} \mid \textit{Serial} \mid \textit{Threads} \mid \textit{OpenMP} \mid \textit{Cuda} \mid \dots \\
\mu &\in \textit{Memory Space} ::= \textit{Host} \mid \textit{CudaUVM} \mid \textit{Cuda} \mid \dots \\
E &\in \textit{Expression} ::= x \mid x(E) \mid c \mid E_0 \textit{ op}_i E_1 \\
S &\in \textit{Statement} ::= C; S \mid \textit{ret} \mid \textit{decl } x := E; S \mid \textit{decl } x \textit{ in } \mu; S \\
C &\in \textit{Command} ::= \textit{set } x := E \mid \textit{set } x(E_0) := E_1 \\
&\quad \mid \textit{fence}(\chi) \mid \textit{deep\_copy}(E_0, E_1) \mid \textit{kernel}(\chi, \lambda x_0, \dots, x_n. S)
\end{aligned}$$
**Fig. 1.** H-IMP Syntax.

- A type system that provides static checks on the spaces for both computations and memory (Sect. 4). Our appendix contains the detailed proof that well-typed programs are free of heterogeneous memory errors by means of a realizability model of the type system over its operational semantics.<sup>1</sup>
- An extension to H-IMP to include variables, like *default*, for execution and memory spaces. Thereby, we can give a *concise, formal* definition of what it means for a program to be portable to other architectures (Sect. 5). Moreover, our space variables allow us to write programs for portable, multiple-accelerator nodes that are currently not expressible in the Kokkos library.

## 2 A Syntax for Computing with Accelerators

Figure 1 presents the syntax of H-IMP. The language is a heterogeneous modification of the language IMP, a common model for imperative languages [16]; similarly, we construct programs from statements which consist of commands and expressions. Whereas commands are used to modify program state imperatively, expressions compute pure values from the program state. To model the heterogeneity in a similar manner to Kokkos, H-IMP has execution and memory spaces. Execution spaces, denoted  $\chi$ , are more general than mere devices; *e.g.* OpenMP is an execution space but may run on CPUs or accelerators. Similarly, several different kinds of memory spaces, denoted  $\mu$ , can exist on the same device; each with different characteristics. For instance, some memory spaces, like *CudaUVM*, are accessible from multiple execution spaces.

Though we specify a number of execution and memory spaces in Fig. 1, these are not intended to be fixed sets, which is why they are written with ellipses. In later sections, we will see how one can expand and contract these sets as well as describe their accessibility properties to influence the strength our portability theorem for a specific program.

The imperative features of H-IMP are for mutating variables and views as well as launching kernels. There are two kinds of commands for declaring local variables: the first declares a local variable for an expression and the second declares a view in memory space  $\mu$  while binding a pointer to it locally. Here, we require that a view declaration include an explicit memory space where its data is allocated; this is a necessary intermediate step to describing portable

<sup>1</sup> The paper with appendix is available at <https://proof.sandia.gov/#himp24>.

$GState \in$	<i>Global State</i>	$::= \langle \mathbb{M} \parallel \mathbb{S} \parallel L \parallel S \rangle$
$LState \in$	<i>Local State</i>	$::= \langle \mathbb{M} \parallel L \parallel S \rangle$
$Conf \in$	<i>Expr. Conf.</i>	$::= \langle \mathbb{M} \parallel L \parallel E \rangle$
$\mathbb{M} \in$	<i>Mach. Mem. Spaces</i>	$= Mem. Space \rightarrow Pointer \rightarrow \mathbb{N} \rightarrow B_i$
$L \in$	<i>Local Mem.</i>	$= Variable \rightarrow Mach. Value$
$V, W \in$	<i>Mach. Value</i>	$::= (\mu, \pi) \mid c$
$\mathbb{S} \in$	<i>Ex. Space Queues</i>	$= FIFO^{Ex. Space} (Local Mem. \times Statement)$

**Fig. 2.** Operational Semantics Syntax.

programs with *default* spaces in Sect. 5. Similar to the two kinds of variable access, we have two different notions of mutating variables: those for views and non-views. Other commands available are those for synchronizing the host with a particular execution space, copying between memory spaces, and launching new kernels on a particular execution space. The notion of a kernel in H-IMP is the more general than those found in Kokkos; that is, our kernels consist only of a particular execution space in which they execute and a set of variables they copy from the host into the runtime environment (which may itself be the host).

Excluded from this study are loops and conditionals because we focus on the features related to portability of heterogeneous systems, *i.e.* those which launch and control kernels of different accelerators while communicating through shared variables. We include two key features that enable communication and synchronization: deep copy and fence, respectively. Deep copy enables the movement of data between views, while a fence blocks until completion of all asynchronous operations. We do include constants  $c$  from a set of base types  $B_i$  and operations over them  $E_0$  *op*  $E_1$  for use in our examples. Indexing into views is done with natural numbers, which are an example of these constants for the base type  $\mathbb{N}$ .

### 3 Operational Semantics

The goal of operational semantics is to model the concurrent execution of kernels from different accelerators alongside a collection of memory spaces within an abstract machine. The syntax for it is found in Fig. 2. It contains three different kinds of program state for which we define three different notions of evaluation. All states contain a local environment  $L$  that contain local variables. The largest state, *i.e.* global state, has access to all of the memory spaces available, written  $\mathbb{M}$ , as well as the queues of work for the execution spaces available, written  $\mathbb{S}$ . Local states are for kernel execution and consist of local memory, one statement for execution, and a restricted set of available memory spaces. Local states cannot access any work queues for execution spaces. Expression evaluation configurations contain the same information.

The available memory spaces ( $\mathbb{M}$ ) are a partial map from memory spaces to pointers to indices to values of base types (such as integers). To access a specific index  $n$  of a view  $\pi$  in a memory space  $\mu$ , we write  $\mathbb{M}(\mu)(\pi)(n)$ ; if we just wanted the particular view, then we would write  $\mathbb{M}(\mu)(\pi)$ ; and so on. For simplicity, we

$$\boxed{\langle\langle \mathbb{M} \models L \parallel E \rangle\rangle \Downarrow V}$$

$$\begin{array}{c}
\frac{x \in \text{Dom}(L)}{\langle\langle \mathbb{M} \models L \parallel x \rangle\rangle \Downarrow L(x)} \text{EVar} \quad \frac{}{\langle\langle \mathbb{M} \models L \parallel c \rangle\rangle \Downarrow c} \text{EConst} \\
\frac{\langle\langle \mathbb{M} \models L \parallel E \rangle\rangle \Downarrow n \quad L(x) = (\mu, \pi) \quad \pi \in \text{Dom}(\mathbb{M}(\mu))}{\langle\langle \mathbb{M} \models L \parallel x(E) \rangle\rangle \Downarrow \mathbb{M}(\mu)(\pi)(n)} \text{EViewDeref} \\
\frac{\langle\langle \mathbb{M} \models L \parallel E_0 \rangle\rangle \Downarrow c_0 \quad \langle\langle \mathbb{M} \models L \parallel E_1 \rangle\rangle \Downarrow c_1}{\langle\langle \mathbb{M} \models L \parallel E_0 \text{ op } E_1 \rangle\rangle \Downarrow c_0 \text{ op } c_1} \text{EOp}
\end{array}$$

**Fig. 3.** Expression Evaluations

assume that if a view is defined then any index into it is defined. This syntax follows similarly for local memory, but there fewer levels of indirection; that is, we need only write  $L(x)$ . Additionally, whereas views can only contain values of base types  $c$ , local memory can contain both values of base types and pointers to views in memory  $(\mu, \pi)$ . We use the syntax  $L[x \mapsto V]$  to denote either replacing the current mapping of  $x$  in  $L$  or to insert a new mapping for  $x$  when it does not yet exist in  $L$ . Likewise, we can update our available memory spaces;  $\mathbb{M}[\mu, \pi, n \mapsto c]$  updates (or inserts)  $c$  at the  $n$ th index of the view at location  $\pi$  in memory space  $\mu$ . We use  $\text{Dom}$  (short for domain) to ensure variables exist in their respective environments (local or memory spaces). Finally, we use  $\mathbb{M}|_{P(\mu)}$  to denote the restriction of the memory spaces to those in the set of memory spaces  $\mu$  that satisfy the proposition  $P$ .

The execution space queues, denoted  $\mathbb{S}$ , contained within the global state is an execution-space-indexed first-in-first-out queue. All of the operations on this object include a specific memory space.  $\mathbb{S}.\text{empty}(\chi)$  is a proposition that is true if the work queue for execution space  $\chi$  is empty.  $\mathbb{S}.\text{pusht}(\chi, (L, S))$  publishes a new task to the end of  $\chi$ 's work queue.  $\mathbb{S}.\text{head}(\chi)$  merely looks at the front of the queue; whereas  $\mathbb{S}.\text{poph}(\chi)$  removes the front of the queue. Finally, we have  $\mathbb{S}.\text{replaceh}(\chi, (L, S))$  which updates the head of the queue to a new work state.

We first present big-step reduction of expression configurations to machine values in Fig. 3. For variables, we merely look it up in the local memory. For accessing views, we first evaluate the index with the current state to get a pointer to a particular view in a memory space, and then we index into  $\mathbb{M}$  with it. If the view that we are trying to dereference is not in the local  $\mathbb{M}$  then we would not be able to construct an evaluation derivation. In a real program, this would occur if the current execution space does not have access to that memory space, since it would not be included in local instance of  $\mathbb{M}$ . Such a restriction is upheld when instantiating a kernel by the *GXStep* rule in Fig. 5.

Taking steps locally, which includes execution spaces transitioning, is defined by the deterministic relation in Fig. 4. Declaring and mutating variables both happen by evaluating the expression and using the result to manipulate the local environment  $L$ . Of course, failing to declare a local variable before setting it will result in a local memory error, so no transition is possible. We may

$$\boxed{\langle\langle \mathbb{M} \models L \parallel S \rangle\rangle \mapsto \langle\langle \mathbb{M}' \models L' \parallel S' \rangle\rangle}$$

$$\frac{\langle\langle \mathbb{M} \models L \parallel E \rangle\rangle \Downarrow V}{\langle\langle \mathbb{M} \models L \parallel \text{decl } x := E; S \rangle\rangle \mapsto \langle\langle \mathbb{M} \models L[x \mapsto V] \parallel S \rangle\rangle} \text{ } L\text{DeclVar}$$

$$\frac{x \in \text{Dom}(L) \quad \langle\langle \mathbb{M} \models L \parallel E \rangle\rangle \Downarrow V}{\langle\langle \mathbb{M} \models L \parallel \text{set } x := E; S \rangle\rangle \mapsto \langle\langle \mathbb{M} \models L[x \mapsto V] \parallel S \rangle\rangle} \text{ } L\text{SetVar}$$

$$\frac{\begin{array}{c} L(x) = (\mu, \pi) \quad \pi \in \text{Dom}(\mathbb{M}(\mu)) \\ \langle\langle \mathbb{M} \models L \parallel E_0 \rangle\rangle \Downarrow n \quad \langle\langle \mathbb{M} \models L \parallel E_1 \rangle\rangle \Downarrow c \end{array}}{\langle\langle \mathbb{M} \models L \parallel \text{set } x(E_0) := E_1; S \rangle\rangle \mapsto \langle\langle \mathbb{M}[\mu, \pi, n \mapsto c] \models L \parallel S \rangle\rangle} \text{ } L\text{SetView}$$

**Fig. 4.** Local Transitions.

also mutate views from execution spaces, which has a similar restriction that the location must be already defined before changing it. Like with the big-step rule for expressions,  $\mathbb{M}$  may or may not contain the particular memory spaces and views to complete a transition depending on the instantiation of the kernel. Finally, local transitions operate over statements, but do not have the permission to allocate new views, deep copy, or fence; thus, such statements would be stuck.

Global transitions are described in Fig. 5. Intuitively, these transitions represent the host program, which orchestrates all of the memory and execution spaces. The first rule *GHStep* is for when the host takes a step locally in the same manner as an execution space. Unlike other execution spaces, the host can also declare a view, with *GDeclView*, given an unused view location  $\pi$ . We take  $\mathbb{M}[\mu, \pi \mapsto \text{init}]$  to mean that for any  $n$  that  $\mathbb{M}(\mu)(\pi)(n)$  is defined. In *GKernel*, the host program publishes a new unit of work to an execution space's work stack; note that it also copies the local variables captured by the  $\lambda$ -expression in the kernel definition, which can get stuck if the variables are undefined. In *GFence*, we see that the program is stuck until the execution space, for which we are waiting, completes its stack of work. Finishing a unit of work in the stack is achieved by the two concurrent steps of *GXPpop* and *GXStep*. The first removes the work when **ret** is the waiting statement. The latter selects one of the execution space queues and takes a single step on it. It is in this rule that we restrict the valid memory spaces for each execution space when it takes a step; we use  $\mu \triangleright \chi$  for the restricted set of memory spaces  $\mu$  that are accessible from  $\chi$ . Note that because global transitions are non-deterministic, these steps model the concurrency implicit in the Kokkos machine model.

The following definitions describe how to run programs in our abstract machine; we will later define *safe* executions of the machine and our static analysis will show well-typed H-IMP programs imply safe execution (Theorem 1).

**Definition 1 (Initial State).**  $\text{Initial}(\langle\langle \mathbb{M} \parallel \mathbb{S} \models L \parallel S \rangle\rangle)$  where  $\mathbb{M}$  contains a set of empty memory spaces and  $\mathbb{S}$  contains all empty work stacks.

**Definition 2 (Final States).** For local states,  $\text{Final}(\langle\langle \mathbb{M} \models L \parallel \text{ret} \rangle\rangle)$ .



$$\boxed{\langle\langle \mathbb{M} \parallel \mathbb{S} \models L \parallel S \rangle\rangle \longrightarrow \langle\langle \mathbb{M}' \parallel \mathbb{S}' \models L' \parallel S' \rangle\rangle}$$

$$\frac{C \in \{\text{decl } x := E, \text{set } x := E, \text{set } x(E_0) := E_1\} \quad \langle\langle \mathbb{M} \models L \parallel C; S \rangle\rangle \mapsto \langle\langle \mathbb{M}' \models L' \parallel S \rangle\rangle}{\langle\langle \mathbb{M} \parallel \mathbb{S} \models L \parallel C; S \rangle\rangle \longrightarrow \langle\langle \mathbb{M}' \parallel \mathbb{S} \models L' \parallel S \rangle\rangle} \text{GHStep}$$

$$\frac{\pi \notin \text{Dom}(\mathbb{M}(\mu))}{\langle\langle \mathbb{M} \parallel \mathbb{S} \models L \parallel \text{decl } x \text{ in } \mu; S \rangle\rangle \longrightarrow \langle\langle \mathbb{M}[\mu, \pi \mapsto \text{init}] \parallel \mathbb{S} \models L[x \mapsto (\mu, \pi)] \parallel S \rangle\rangle} \text{GDeclView}$$

$$\frac{\forall i \in 0, \dots, n. x_i \in \text{Dom}(L) \quad L' = x_0 \mapsto L(x_0), \dots, x_n \mapsto L(x_n)}{\langle\langle \mathbb{M} \parallel \mathbb{S} \models L \parallel \text{kernel}(\chi, \lambda x_0, \dots, x_n. S_0); S_1 \rangle\rangle \longrightarrow \langle\langle \mathbb{M} \parallel \mathbb{S}. \text{pusht}(\chi, (L', S_0)) \models L \parallel S_1 \rangle\rangle} \text{GKernel}$$

$$\frac{\mathbb{S}. \text{empty}(\chi)}{\langle\langle \mathbb{M} \parallel \mathbb{S} \models L \parallel \text{fence}(\chi); S \rangle\rangle \longrightarrow \langle\langle \mathbb{M} \parallel \mathbb{S} \models L \parallel S \rangle\rangle} \text{GFence}$$

$$\frac{\langle\langle \mathbb{M} \models L \parallel E_0 \rangle\rangle \Downarrow (\mu_0, \pi_0) \quad \langle\langle \mathbb{M} \models L \parallel E_1 \rangle\rangle \Downarrow (\mu_1, \pi_1)}{\langle\langle \mathbb{M} \parallel \mathbb{S} \models L \parallel \text{deep\_copy}(E_0, E_1); S \rangle\rangle \longrightarrow \langle\langle \mathbb{M}[\mu_1, \pi_1 \mapsto \mathbb{M}(\mu_0)(\pi_0)] \parallel \mathbb{S} \models L \parallel S \rangle\rangle} \text{GDeepCopy}$$

$$\frac{\mathbb{S}. \text{head}(\chi) = (L', \text{ret})}{\langle\langle \mathbb{M} \parallel \mathbb{S} \models L \parallel S \rangle\rangle \longrightarrow \langle\langle \mathbb{M} \parallel \mathbb{S}. \text{poph}(\chi) \models L \parallel S \rangle\rangle} \text{GXPop}$$

$$\frac{\mathbb{S}. \text{head}(\chi) = (L', S') \quad \langle\langle \mathbb{M}[\mu \triangleright \chi] \models L' \parallel S' \rangle\rangle \mapsto \langle\langle \mathbb{M}' \models L'' \parallel S'' \rangle\rangle}{\langle\langle \mathbb{M} \parallel \mathbb{S} \models L \parallel S \rangle\rangle \longrightarrow \langle\langle \mathbb{M}[\mu \triangleright \chi] \cup \mathbb{M}' \parallel \mathbb{S}. \text{replaceh}(\chi, (L', S'')) \models L \parallel S \rangle\rangle} \text{GXStep}$$

**Fig. 5.** Global Transitions.

$$\begin{aligned}
\tau &\in \text{Type} & ::= B_i \mid \text{view}(\mu, B_i) \\
\Gamma &\in \text{Type Environment} & ::= \varepsilon \mid \Gamma, x:\tau
\end{aligned}$$

**Fig. 6.** H-IMP Type Syntax.

For global states,  $\text{Final}(\langle\langle \mathbb{M} \parallel \mathbb{S} \models L \parallel \text{ret} \rangle\rangle)$  where  $\mathbb{S}$  contains all empty work stacks.

## 4 Type System

We present the syntax of the H-IMP type system in Fig. 6. Memory spaces are referenced by  $\text{view}(\mu, B_i)$  types, which are pointers to data structures over the base type  $B_i$  and housed within a memory space  $\mu$ . As a simplification from Kokkos, we consider views to be arrays of type  $B_i$ .

To reason statically about memory spaces and execution spaces of H-IMP programs, the judgements of our type system require information about where their computations occur and the information about the memory spaces accessible from each execution space must be supplied. The type system's rules are presented in Fig. 7. There are three main judgements that all end in  $@ \chi$  signifying the execution space wherein the expression, statement, or command is to take place. For instance,  $\Gamma \vdash E : \tau @ \chi$  states that with the local type environment  $\Gamma$  the expression  $E$  computes a value of type  $\tau$  in the execution

$$\boxed{\Gamma \vdash E : \tau @ \chi}$$

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau @ \chi} TVar \quad \frac{c \in B_i}{\Gamma \vdash c : B_i @ \chi} TConst$$

$$\frac{x:\mathbf{view}(\mu, B_i) \in \Gamma \quad \mu \triangleright \chi \quad \Gamma \vdash E : \mathbb{N} @ \chi}{\Gamma \vdash x(E) : B_i @ \chi} TViewDeref$$

$$\frac{\Gamma \vdash E_0 : \tau_0 @ \chi \quad \Gamma \vdash E_1 : \tau_1 @ \chi \quad op_i : \tau_0 \rightarrow \tau_1 \rightarrow \tau}{\Gamma \vdash E_0 \ op_i \ E_1 : \tau @ \chi} TOp$$

$$\boxed{\Gamma \vdash S @ \chi}$$

$$\frac{\Gamma \vdash C @ \chi \quad \Gamma \vdash S @ \chi}{\Gamma \vdash C; S @ \chi} TCom \quad \frac{}{\Gamma \vdash \mathbf{ret} @ \chi} TRet$$

$$\frac{\Gamma \vdash E : \tau @ \chi \quad \Gamma, x:\tau \vdash S @ \chi}{\Gamma \vdash \mathbf{decl} \ x := E; S @ \chi} TDeclVar \quad \frac{\Gamma, x:\mathbf{view}(\mu, B_i) \vdash S @ \mathbf{Host}}{\Gamma \vdash \mathbf{decl} \ x \ \mathbf{in} \ \mu; S @ \mathbf{Host}} TDeclView$$

$$\boxed{\Gamma \vdash C @ \chi}$$

$$\frac{x:\tau \in \Gamma \quad \Gamma \vdash E : \tau @ \chi}{\Gamma \vdash \mathbf{set} \ x := E @ \chi} TSetVar \quad \frac{}{\Gamma \vdash \mathbf{fence}(\chi) @ \mathbf{Host}} TFence$$

$$\frac{x:\mathbf{view}(\mu, B_i) \in \Gamma \quad \mu \triangleright \chi \quad \Gamma \vdash E_0 : \mathbb{N} @ \chi \quad \Gamma \vdash E_1 : B_i @ \chi}{\Gamma \vdash \mathbf{set} \ x(E_0) := E_1 @ \chi} TSetView$$

$$\frac{\Gamma \vdash E_0 : \mathbf{view}(\mu_0, B_i) @ \mathbf{Host} \quad \Gamma \vdash E_1 : \mathbf{view}(\mu_1, B_i) @ \mathbf{Host}}{\Gamma \vdash \mathbf{deep\_copy}(E_0, E_1) @ \mathbf{Host}} TDeepCopy$$

$$\frac{\forall i \in 0, \dots, n. x_i:\tau_i \in \Gamma \quad \chi \neq \mathbf{Host} \quad x_0:\tau_0, \dots, x_n:\tau_n \vdash S @ \chi}{\Gamma \vdash \mathbf{kernel}(\chi, \lambda x_0, \dots, x_n. S) @ \mathbf{Host}} TKernel$$

**Fig. 7.** H-IMP Typing Rules.

space  $\chi$ . Certain commands are only available to the host execution space, the orchestrator of H-IMP programs. Specifically, the host is the only execution space that may declare views, fence execution spaces, deep copy views, and launch kernels. However, we cannot launch kernels for the host; one would instead need to use the **Serial** execution space.

Note that the typing environment  $\Gamma$  will *only* contain variables local to that execution space. During computation this is thread-local memory; see that the *TKernel* rule specifies explicitly the variables that will be copied to its local memory.

Indexed by some sets of memory and execution spaces, our typing system depends on a relation  $\mu \triangleright \chi$  on *Mem. Space*  $\times$  *Ex. Space*, which occurred in the operational semantics. For the set of execution and memory spaces we gave in Fig. 1, this relation is defined as the following:

$$(\triangleright) = \{(\text{Host}, \text{Host}), (\text{Host}, \text{Serial}), (\text{Host}, \text{Threads}), (\text{Host}, \text{OpenMP}), (\text{Cuda}, \text{Cuda})\} \\ \cup \{(\text{CudaUVM}, \chi) \mid \chi \in \text{Ex. Space}\}$$

In this relation, `CudaUVM` can safely be accessed by any execution space  $\chi$ ; of course, this may not be true if we wanted to consider GPUs from another vendor. The rules *TViewDeref* and *TSetView* check that every view referenced is accessible to the current execution space.

#### 4.1 Safety

We must define a notion safety for each class of computable syntax. Expression configurations are the simplest: they are safe if they evaluate to a machine value. Both global and local machine states are safe if they take any number of steps to either a final state or they can continue to step; *i.e.* they cannot reach a stuck state.

**Definition 3 (Safe Configurations and States).** *For an expression configuration,  $\text{Safe}(\text{Conf})$  if and only if  $\text{Conf} \Downarrow V$ .*

*For an execution-space state,  $\text{Safe}(X\text{State})$  if and only if  $X\text{State} \mapsto^* X\text{State}'$  implies  $\text{Final}(X\text{State}')$  or  $X\text{State}' \mapsto X\text{State}''$ .*

*For a host state,  $\text{Safe}(G\text{State})$  if and only if  $G\text{State} \longrightarrow^* G\text{State}'$  implies  $\text{Final}(G\text{State}')$  or  $G\text{State}' \longrightarrow G\text{State}''$ .*

Though this looks like an overly simple notion of safety, it implies that we are always accessing an accessible view from the current execution space, that the global state is only manipulated directly by host execution space, and that variables are initialized before they are mutated. Moreover, it even captures safety in the notion of concurrency employed by the global transitions; because for every way that we take a step—there are multiple—we must step to a good final state or keep stepping.

**Theorem 1 (Type Safety).** *If  $\vdash S @ \text{Host}$ , then  $\text{Safe}(\text{Init}(S))$ .*

## 5 Portable Programs

Kokkos programs, and templated C++ programs more generally, require abstracted template variables to be instantiated with concrete types and functions before a complete binary can be run. The programming language that we just presented can be seen as a program where all of the decisions about a node’s architecture have been decided. However, this is not how the Kokkos C++ library is intended to be used. We would like H-IMP instead to specify one program that works for many different architectures. To accomplish this, Kokkos programs do not need to specify explicitly the memory spaces wherein

views are located, or the execution spaces whereat kernels are executed. Thus, we may see a source program (in H-IMP) like the following:

```
decl x in defaultMem;
kernel(defaultEx, λx. x(0) := 2; ret);
...
```

Before running this program with our machine machine, we must decide how these default spaces are instantiated. If a program is portable, then it should be the case that *any* instantiation of the default spaces produces a safe program.

**Definition 4 (Portable Program).** *A program  $S$  is portable if and only if  $\text{Safe}(\text{Init}(S[\sigma]))$  for a given set of execution and memory spaces, their accessibility relation, and any instantiation  $\sigma$  of its free execution and memory variables.*

To describe this “templated” H-IMP, we must add memory and execution space variables to programs, denoted with underlines:

$$\begin{aligned} \chi &\in \text{Ex. Space} && ::= x \mid \underline{\chi} \\ \underline{\chi} &\in \text{Inst. Ex. Space} && ::= \text{Host} \mid \text{Threads} \mid \text{OpenMP} \mid \text{Cuda} \mid \dots \\ \mu &\in \text{Mem. Space} && ::= x \mid \underline{\mu} \\ \underline{\mu} &\in \text{Inst. Mem. Space} && ::= \text{Host} \mid \text{CudaUVM} \mid \text{Cuda} \mid \dots \\ \Delta &\in \text{Space Env.} && ::= \varepsilon \mid \Delta, \text{ex } x \mid \Delta, \text{mem } x \end{aligned}$$

In real Kokkos programs, the *default* memory and execution space variables exist implicitly, but only as special space variables. Here, we have the option of multiple default spaces; consider for instance, a program with *defaultEx1* and *defaultEx2*, which could be instantiated on several kinds of two GPU systems.

To statically reason about space variables, we extend our typing judgments with  $\Delta$  containing the free memory and execution space variables. For example, our expression judgments would have the form  $\Gamma \vdash_{\Delta} E : \tau @ \chi$ . The accessibility relation now only refers to fully instantiated memory spaces  $\underline{\mu} \supseteq \underline{\chi}$ , and we have a new generalized relation with the judgement  $\Delta \vdash \mu \triangleright \chi$  that we use in the updated rules from Fig. 7.<sup>2</sup>

$$\frac{\Delta \vdash \mu :: \text{MS} \quad \Delta \vdash \chi :: \text{ES} \quad \forall \sigma \in \text{Inst}(\Delta), \mu[\sigma] \supseteq \chi[\sigma]}{\Delta \vdash \mu \triangleright \chi}$$

The judgements  $\Delta \vdash \mu :: \text{MS}$  and  $\Delta \vdash \chi :: \text{ES}$  are added to check that a space is either a variable in  $\Delta$  or an instance. This extended type system is strong enough to give us portability.

**Theorem 2 (Typing Ensures Portability).** *If  $\Gamma \vdash_{\Delta} S @ \text{Host}$ , then  $S$  is portable.*

<sup>2</sup> The full type system with the new and rewritten rules is found in the appendix.

Note well that constructing portable programs is very limited. We cannot prove the generalized accessibility rule unless we show that for any combination memory and execution space that they are accessible. For example, we cannot prove  $\text{mem default} \vdash \text{default} \triangleright \text{Cuda}$ , because there exists a memory space inaccessible to Cuda execution spaces: Host. Indeed, given the set of execution and memory spaces of Fig. 1 and the relation specified in Sect. 4 there exist *no* portable programs that make use of views with *default* memory and execution spaces. Thus, specifying a portable program necessarily includes giving a restricted set of execution and memory spaces for which it is portable.

## 6 Conclusion

We have developed a language H-IMP as a distillation of the features for portable heterogeneity present in Kokkos wherein we can launch kernels for different accelerators. An important notion is that of the permissions for each execution space, which controls the different types of memory it can access and the operations that it may perform. Over this language, we defined a type system that allows us to guarantee that well-typed programs do not misuse heterogeneous memory. Finally, we defined a notion of portable programs for this language and noted that there are no meaningful portable programs without specifying the restricted set of architectures to which a program is portable.

As future work, we plan to enhance the language with the typeclass mechanism [14] found in Haskell. This will allow us to describe portable programs as those that can be run on *any* architecture that satisfy some constraint, thereby avoiding the proviso that we specify a specific set of spaces. For instance, kernel code could have the type  $\Gamma \vdash_{\Delta} S @ \forall \chi. \text{Host} \triangleright \chi \Rightarrow \chi$  meaning that it can run in any execution space that can access host memory. In addition, we imagine an extension of the types to include more detailed information about the architecture including types representing the parallelism hierarchy of certain execution spaces and the interaction of kernels with different memory models. Currently, we are developing a tool for real Kokkos programs that uses this reasoning to identify the sets of architectures for which a program is portable.

**Acknowledgment.** Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-NA0003525.

Additionally, we thank Jackson Mayo, Keita Teranishi, Christian Trott, Vivek Kale, Shyamali Mukherjee, Richard Rutledge, John Bender, and our anonymous reviewer for comments on draft versions of this paper.

## References

1. Ahmed, A., Jia, L., Walker, D.: Reasoning about hierarchical storage. In: 18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings, pp. 33–44 (2003). <https://doi.org/10.1109/LICS.2003.1210043>

2. Beckingsale, D.A., et al.: RAJA: Portable performance for large-scale scientific applications. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 71–81 (2019). <https://doi.org/10.1109/P3HPC49587.2019.00012>
3. Callahan, D., Chamberlain, B.L., Zima, H.P.: The cascade high productivity language. In: 9th International Workshop on High-Level Programming Models and Supportive Environments (HIPS 2004), 26 April 2004, Santa Fe, NM, USA, pp. 52–60. IEEE Computer Society (2004). <https://doi.org/10.1109/HIPS.2004.1299190>
4. Charles, P., et al.: X10: an object-oriented approach to non-uniform cluster computing. In: Johnson, R.E., Gabriel, R.P. (eds.) Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16–20, 2005, San Diego, CA, USA, pp. 519–538. ACM (2005). <https://doi.org/10.1145/1103845.1094852>
5. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998). <https://doi.org/10.1109/99.660313>
6. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). <https://doi.org/10.1016/j.jpdc.2014.07.003>
7. Jin, F., Jacobson, J., Pollard, S.D., Sarkar, V.: Minikokkos: a calculus of portable parallelism. In: Laguna, I., Rubio-González, C. (eds.) Sixth IEEE/ACM International Workshop on Software Correctness for HPC Applications, Correctness@SC 2022, Dallas, TX, USA, November 13–18, 2022, pp. 37–44. IEEE (2022). <https://doi.org/10.1109/Correctness56720.2022.00010>
8. Khronos SYCL Working Group: Sycl 2020 specification (revision 8) (2020). <https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf>
9. Lee, J.K., Palsberg, J.: Featherweight x10: a core calculus for async-finish parallelism. SIGPLAN Not. **45**(5), 25–36 (Jan 2010). <https://doi.org/10.1145/1837853.1693459>
10. Rossbach, C.J., Yu, Y., Currey, J., Martin, J., Fetterly, D.: Dandelion: a compiler and runtime for heterogeneous systems. In: Kaminsky, M., Dahlin, M. (eds.) ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3–6, 2013, pp. 49–68. ACM (2013). <https://doi.org/10.1145/2517349.252271>
11. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas Commun. **21**(1), 5–19 (2003). <https://doi.org/10.1109/JSAC.2002.806121>
12. Tofte, M., Talpin, J.: Region-based memory management. Inf. Comput. **132**(2), 109–176 (1997). <https://doi.org/10.1006/inco.1996.2613>
13. Trott, C.R., et al.: Kokkos 3: programming model extensions for the exascale era. IEEE Trans. Parallel Distrib. Syst. **33**(4), 805–817 (2022). <https://doi.org/10.1109/TPDS.2021.3097283>
14. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 60–76. POPL '89 (1989). <https://doi.org/10.1145/75277.7528>
15. Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC — first experiences with real-world applications. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 859–870. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-32820-6\\_85](https://doi.org/10.1007/978-3-642-32820-6_85)
16. Winskel, G.: The Formal Semantics of Programming Languages - An Introduction. MIT Press, Foundation of computing series (1993)



# Evaluation of Speedup and Energy with Multigrain Parallelizing Compiler

John Pickar<sup>(✉)</sup> , Tohma Kawasumi , Hiroki Mikami, Keiji Kimura ,  
and Hironori Kasahara

Department of Computer Science and Engineering, Waseda University, Green  
Computing Center, 27 Waseda-machi, Shinjuku-ku, Tokyo 162-0042, Japan  
{pickar,tohma,hiroki}@kasahara.cs.waseda.ac.jp,  
{keiji,kasahara}@waseda.jp

**Abstract.** As global computational demand continues to rise, modern multicore architectures play a pivotal role in achieving and providing optimal runtime and energy efficient computing solutions. However, optimizing for both performance and energy efficiency remains a challenge. In addition, developing parallel code and optimizing for different architecture is often time consuming. The OSCAR (Optimally Scheduled Advanced Multiprocessor) compiler, an automatic parallelizing source-to-source compiler, is able to leverage multigrain parallelism to enhance multicore efficiency on a variety of architectures. This allows it to reduce runtime and energy consumption by exploiting parallelism. Furthermore, using data and control dependency analysis in addition to scheduling features, it can apply cache optimization and data localization techniques to further reduce energy consumption by improving runtime. This paper evaluates the OSCAR compiler versus OpenMP in the ability to reduce energy usage by reducing runtime of scientific benchmarks from SPEC2000 and NAS Parallel Benchmarks suites. It will be done on Intel Icelake-SP and AMD Zen-4 16-core processors. Results showed OSCAR providing runtime and total energy improvements compared to OpenMP. Benchmarks such as NAS's CG demonstrated a 10.6x performance increase and 80% energy savings compared to the sequential benchmark on both systems. In comparison to OpenMP at varying equivalent core count, OSCAR provided a 7% to 9% runtime improvement with a 4% to 9% reduction in energy on both systems across benchmarks. The cache optimization and data localization was shown to have provided a 4% runtime improvement and 4% to 7% energy improvement with OSCAR. This was driven by a reduction in L3 cache misses, translating to a runtime and energy improvement. This was achievable at varying core configurations up to the max amount of cores available on the systems.

**Keywords:** multicore · parallelizing compiler · OSCAR · multigrain · green computing · sustainable computing · energy consumption

# 1 Introduction

There is a consistent need for computational power due to the global interest in artificial intelligence, data processing, and general computing. Complex computational tasks often require extended periods of time to execute, coupled with significant associated energy costs. With the focus of Sustainable Computing, there is a significant emphasis on reducing total energy consumption while improving execution times. Current hardware architectures make use of multi-core designs which integrate multiple processing elements (PE), or cores, within a single package. Parallel computing allows for concurrent execution of these PE, significantly increasing processing throughput and efficiency. It is known that reducing runtime results in a reduction of total energy [12, 22]. Furthermore, Dynamic Voltage and Frequency Scaling (DVFS) can also be utilized to reduce power. This paper will focus on reducing total energy through reducing runtime with the OSCAR compiler.

Enabling parallelism and increasing performance in sequential and parallel programs can be achieved with the OSCAR (Optimally Scheduled Advanced Multiprocessor) compiler. It is an automatic parallelizing compiler capable of parallelizing both C, C++, and Fortran programs. This is accomplished by generating sequential code for a specific number of threads, allowing the compiler and the operating system to bind the threads to cores at runtime [8]. With OSCAR's toolset used to analyze and automatically parallelize programs, it also allows for further optimizations such as cache optimization through data localization. This allows for reducing cache misses [6, 10] which is important due to the increasing gap between processor speed and memory latency [17]. The speedup by reducing memory accesses allows for greater speedup resulting in improved energy performance.

This paper will explain and evaluate the OSCAR compiler's automatic multi-grain parallelism and cache optimization techniques. We will observe how parallelizing and cache optimizations can improve runtime and reduce energy consumption. It will also focus on the performance compared to OpenMP at equivalent core count. Previous papers using automatic parallelizing compilers have demonstrated speedup on older architectures [10, 12]. However these previous evaluations have only looked into performance and energy reduction from parallelization compared to sequential or single threaded results; it did not at equivalent core count. Furthermore, previous usage of the OSCAR compiler has been in conjunction with DVFS [15] and was shown reduce power consumption using idle power states [5]. In this case, the evaluation boards were modified to measure the power of the processor chips directly. This paper will instead evaluate how the OSCAR compiler affects runtime and energy without using DVFS, observing energy used by the package, including last level cache and DRAM.

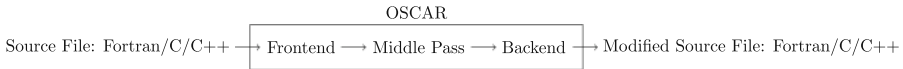
The systems to be evaluated were chosen due to their different cache interconnect design with different memory access latencies [20]; they are an Intel Icelake-SP processor as well as an AMD EPYC Zen-4 processor. Intel uses a mesh network [13] while the AMD has an architecture that is split between core complexes (CCXs) and communicates with an I/O die [18]. The evaluation will



be done by running scientific benchmarks from the NAS parallel benchmark suite in addition to the SPEC benchmark suite.

## 2 The OSCAR Compiler

The OSCAR compiler is an automatic source-to-source parallelizing compiler. It translates provided source code to an intermediary state, performs an analysis and optimizations, then provides optimized source code in the original language provided. Currently Fortran, C, and C++ are supported by the OSCAR Compiler [7]. It can perform fine-grain and coarse-grain parallelization by exploiting loop level parallelism, instruction level parallelism, task based parallelism, and more; this allows it to achieve multigrain parallelism [19]. In order to achieve this automatic multigrain parallelism, the source-to-source compilation is split into multiple steps: frontend, middle pass, and backend (Fig. 1).



**Fig. 1.** OSCAR’s compilation pipeline.

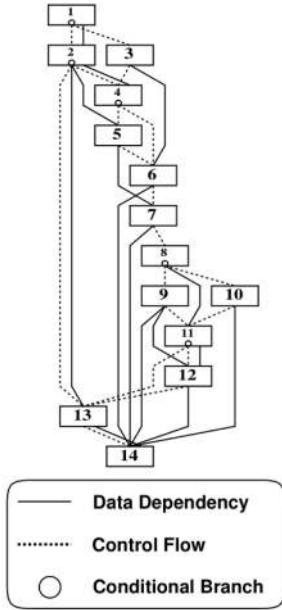
OSCAR’s frontend takes the provided source code and converts it into an intermediate representation. This intermediate representation is the source code broken into various blocks or macro-tasks. Each block or macro-task is a sequence of code that is closely related and executed sequentially. An example of such could be function calls, loops, or assignments [8]. From this state, the middle pass will occur, analyzing the blocks for areas of parallelism and optimization. At this time, the OSCAR compiler will attach a cost to the macro-tasks allowing scheduling to take place [8]. Once the process is complete, the backend will convert the macro-tasks with scheduling information back into the original source language. The resulting source file will contain parallelized code for the number of threads requested, ready to be compiled by any standard compiler (e.g. GNU Compiler Collection, Intel Compiler, LLVM, etc.) [7]. The data localization and cache optimizations performed by OSCAR will occur in the middle pass.

### 2.1 Macro-task Graph

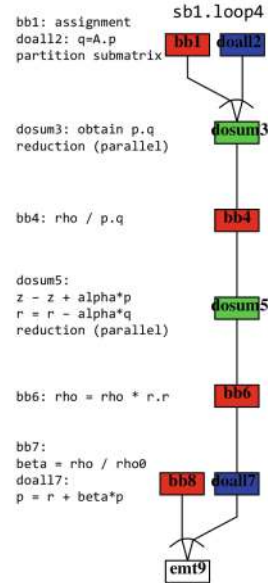
OSCAR’s middle pass is where the majority of the analysis and processing occurs. Once the OSCAR compiler splits the sequential code into macro-tasks, control and data dependency analysis can occur. Using earliest execution analysis, OSCAR can represent and visualize the macro-tasks in a macro-task graph [7]. This macro-task graph is constructed under the conditions that macro-tasks must wait for data it is dependent on to become available in addition to waiting until all prior control-dependencies have been evaluated. As a result, large blocks

of codes such as functions or loops can be represented as a macro-task graph. During this time, a cost is assigned to each macro-task. With this information, a further pass occurs to determine areas where parallelism and optimizations can be applied.

In the macro-task graph seen in Fig. 2a, macro-task 6 cannot be executed until data dependency 3 has completed. Similarly, 14 cannot execute until data and control dependencies have been met. In this example, we can see that parallelism can be exploited starting at 1 and 3 as they have separate data dependencies.



(a) An example macro-task graph showing the control and data dependencies between different macro-tasks [6].



(b) OSCAR's macro-task graph conversion of the main loop of NAS Parallel Benchmark CG from section 4.

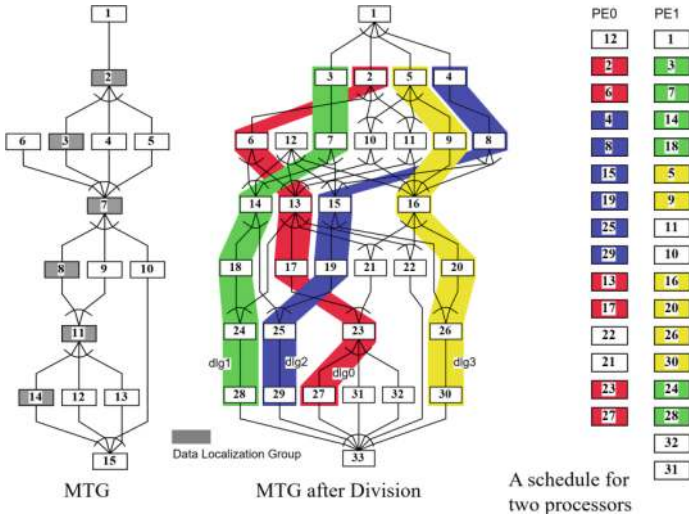
**Fig. 2.** Example macro-task graphs by OSCAR.

Once the source code has been broken into macro-tasks and the corresponding macro-task graphs, scheduling and parallelization can occur. OSCAR will create a thread for each processor element (PE). The macro-tasks are then distributed among the processor elements evenly based on scheduling and cost. OSCAR makes use of static and dynamic scheduling techniques for task-based parallelization. In the case of no control dependencies, static scheduling is used. The macro-task graph is not modified and data dependencies will be managed by barrier spin locks between macro-tasks. As for dynamic scheduling, control

dependencies must be managed. In this case, the macro-task with a control dependency and prior data dependencies may be copied to all processor elements if cost permits so they can be evaluated locally to prevent synchronization overhead [19]. Loop based parallelization can also be applied if a loop has a static number of iterations in addition to the individual loop iterations being independent from other iterations. These cases allow for the loop to be segmented and evenly distributed to the various available processor elements [6].

## 2.2 Cache Optimization and Data Localization

The OSCAR compiler can also apply cache optimization through data localization after the macro-task graphs are generated. It does this using loop-aligned decomposition, loop level parallelism, and the ensuing data localization groups [21]. These data localization groups are defined within specified cache limit sizes to ensure data elements are close to the PE being utilized. Furthermore, it keeps the data localization groups consistent across a series of macro-tasks such as in Fig. 3, reducing cache miss and improving cache coherency [6].



**Fig. 3.** OSCAR applying Loop Aligned Decomposition (LAD) on macro-tasks and dividing them into Data Localization Groups (DLG) [7].

This optimization occurs when macro-tasks are directly connected to doall-loops, reduction-loops, and sequential-loop type macro-tasks with data dependencies only from the proceeding macro-task. For Fig. 2b, this would be applied to the portion of NAS Parallel Benchmark CG's main loop. The groups are constructed to fit within the given cache size of a processing element, then they

are then assigned to processor elements to be executed concurrently while maintaining minimum synchronization from data sharing [6]. If the number of groups exceeds the number of PE, multiple groups may be assigned to a specific PE.

### 2.3 Cache Data Distribution Using First Touch and Thread Binding

On modern computing systems and operating systems, there is a First Touch Placement Policy that allocates the data page to the closest thread accessing the page for the first time. This causes the first single thread or node to allocate all the data, greatly increasing memory latency for threads and congestion for the memory controller [9]. The OSCAR compiler allows for a First Touch policy to manage initial data locality among the processor elements (PE). As memory affinity is defined at initialization, OSCAR uses the threads it generates per PE at runtime to access the data on the PE it is to be localized on before beginning the main program. The threads are bound to the PE and do not migrate during runtime. This allows OSCAR to place data on the correct PE before it begins execution, minimizing memory accesses and memory controller congestion.

## 3 System Architecture

Two different x86-64 processors with different memory hierarchies are being investigated in this paper. One is the Intel Xeon Gold 6326 (Icelake) 16-core processor and the other the AMD EPYC 9124 (Zen-4) 16-core processor (Table 1).

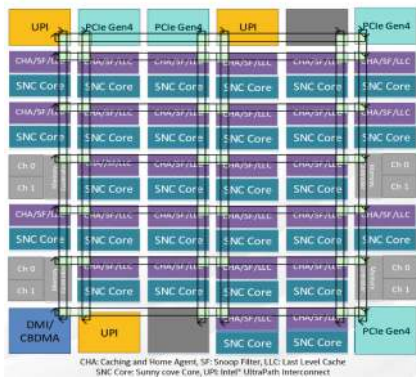
**Table 1.** System Information

System	Cores (PE)	Clock	Boost	L1D	L2	L3 (Shared)	Line Size
Intel	16	3.3 GHz	3.5 GHz	48 KiB	1.25 MiB	24 MiB	64 B
AMD	16	3.0 GHz	3.7 GHz	32 KiB	1.00 MiB	64 MiB	64 B

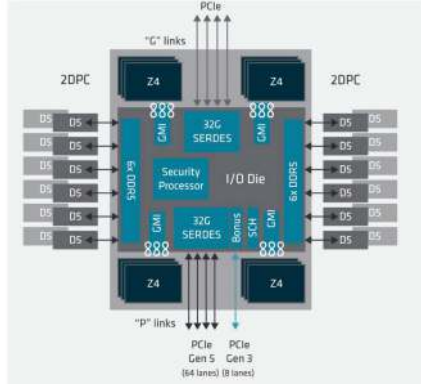
The Intel processor has an all-core frequency of 3.3 GHz with a turboboost of 3.50 GHz. Each core has 1.5 MiB of L3 cache, which is shared among the 16 cores for a total of 24 MiB of shared L3. The shared distributed L3 topology is Intel’s mesh network which is a 2-dimension array of bi-directional half rings forming a system-wide interconnected grid as seen in Fig 4a [11].

The AMD processor has an all-core frequency of 3.0 GHz with a boost of 3.7 GHz. There are 2 Core Complexes (CCX) on this processor which contain 8 cores each. In addition, each CCX contains 32 MiB shared distributed L3 Cache which is connected to the other CCX through data and control fabrics on the I/O die [18].

Due to AMD using multiple CCXs, there is no common shared L3 cache that will always provide the same memory latency. As a result, workloads that



(a) Intel Icelake



(b) AMD Zen4

**Fig. 4.** Intel and AMD SoC [3,13].

require heavy data sharing among cores would benefit from using Intel’s mesh design due to having similar shared L3 latencies between cores [20].

For both processors hyperthreading is disabled meaning there were a total of 16 processing elements (PE) available. Furthermore, the all-core frequencies were used for evaluation. The host systems were running Linux 5.15-generic Kernel or later and using Ubuntu 22.04 LTS. The Intel machine and AMD machine had 236 GiB of DDR4-3200 and 378 GiB of DDR5-4800 RAM respectively. RAPL (Running Average Power Limit) configured by Linux was used to make software energy measurements [2]. The power domain of the entire package was used. The `perf` profile was used to monitor registers related to cache performance.

## 4 Benchmarks

For evaluation of the systems and OSCAR, scientific benchmarks were used. Two different benchmark suites were used for evaluation in this paper.

The first group was SPEC2000 floating-point suite where SWIM and ART were evaluated. SWIM is a weather prediction program computing a shallow-water model written in Fortran. ART (Adaptive Resonance Theory 2) measures neural network training performance and is written in C.

The second group is from the NAS parallel benchmark suite. The C programming language version was used which was developed by the Real World Computing Project and distributed by the HPCS lab of the University of Tsukuba [1]. Within the test suite BT, CG, and SP benchmarks were evaluated. The BT and SP benchmarks were compiled with problem size B and solve synthetic systems of nonlinear PDEs using either Block Tridiagonal (BT) or Scalar Pentadiagonal (SP) [4]. CG was compiled with problem size C and solves the Conjugate Gradient (CG) which estimates the smallest eigenvalue of a large sparse symmetric positive-definite matrix [4].

The source code of the programs were modified to take a RAPL energy measurement immediately before and after the benchmark's timer start and stop function calls in order to track energy consumption.

Three version of the benchmarks were created:

1. **Sequential/OpenMP**: The default provided benchmark. The SPEC suite only provided sequential benchmarks while the NAS parallel benchmarks utilized OpenMP. In the case for OpenMP, to get sequential performance the number of threads was set to 1.
2. **OSCAR**: The Fortran or C source code for the program was passed to the OSCAR compiler and the resulting source file was compiled with `gfortran` or `gcc`.
3. **OSCAR+CacheOpt**: The same as OSCAR with the addition of cache optimization and data localization strategies discussed in Sects. 2.2 and 2.3. For the Cache Optimization strategy, the localized data-dependent groups were configured to the L2 cache size on Intel and the L3 cache size on AMD (using parameters for size, associativity, line size, and shared cores).

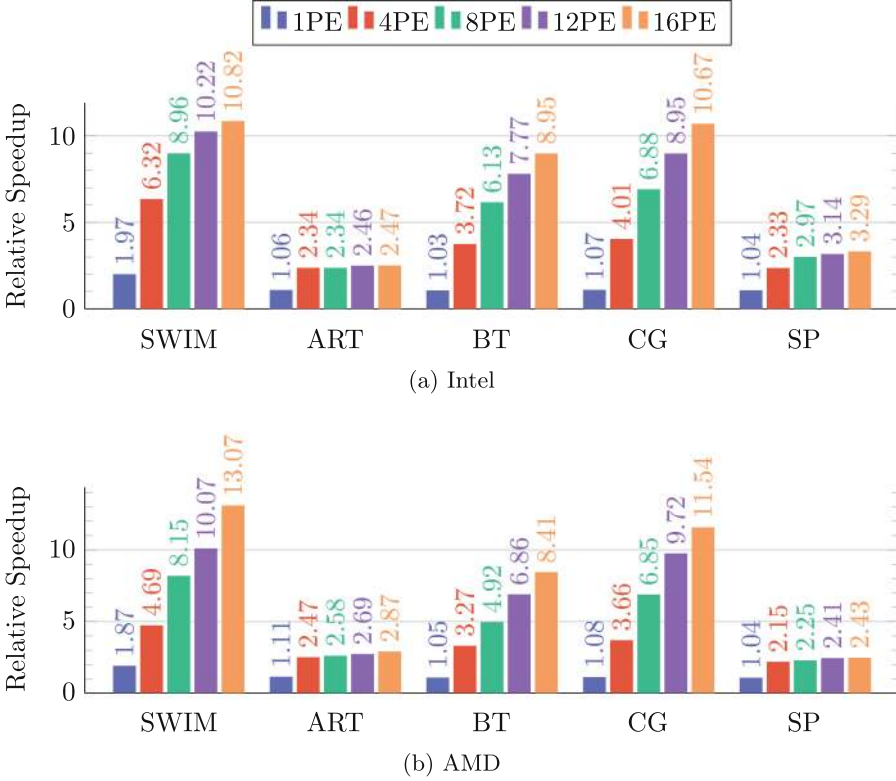
When compiling the benchmarks, the `gcc` or `gfortran` flags used were `-O3` and `-march=native`. The benchmarks were compiled as previously described with no difference between binaries except for OSCAR and OSCAR+CacheOpt having different cache parameters. Thread binding was configured to use `spread`, distributing cores evenly on the system, as it provided the best results. This was applied through `OMP_PROC_BIND` for OpenMP, or manually defined during thread generation for OSCAR. The benchmarks were run 25 times and the median value was for data analysis when summarized.

## 5 Performance Evaluation

Runtime is first looked at on both Intel and AMD systems. In Fig. 5 the speedup of the best automatic parallelization performed by OSCAR is shown. It can be observed that OSCAR is able to achieve superlinear speedup on SWIM for up to 8PE. In addition, BT and CG also achieve scalable results from automatic parallelization. These results can be attributed to OSCAR's scheduling of the macro-tasks, cache optimization, and data localization. However, ART and SP shown saturation with the increase of additional PE. This is due to certain parts of a program that require sequential execution and cannot be easily parallelized with performance improvements due to additional required overhead [14].

### 5.1 Energy

The results can be further segmented looking at the normalized energy values. In Fig. 6, we can see the corresponding energy results. For the benchmarks that scale with additional PE (SWIM, BT, CG), the energy results show a minimum reduction of 77% in energy at 16PE from parallelization. This is inline with the analysis and calculation that optimizing for performance will result in reducing

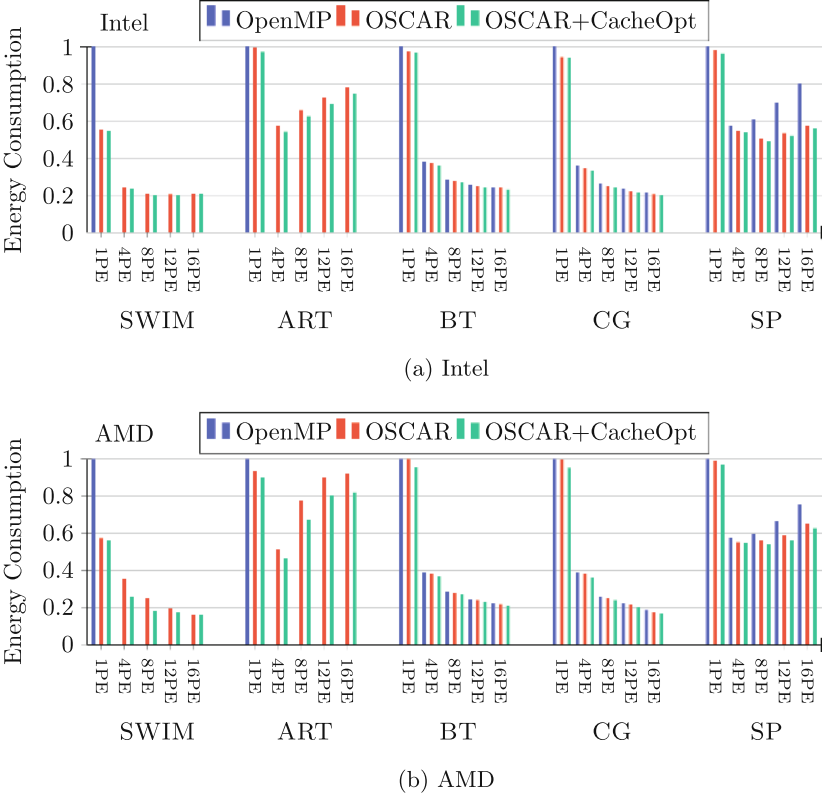


**Fig. 5.** Relative speedup (higher is better) of OSCAR with cache optimizations to base sequential execution (1PE). Data is normalized from values in Table 2.

energy usage [22]. For all three versions of the benchmark, we can see these values present. Furthermore, we can see OSCAR outperforming OpenMP in both runtime and energy with the associated data in Table 2.

As previously mentioned, there is a point where the overhead of parallelization makes it increasingly difficult to improve runtime by simply adding additional PE. With benchmarks ART and SP, the total energy begins to increase after a certain number of PE are used and active. This crossover point is reflected in the decay of runtime improvement. The benchmark can only be optimally parallelized to reduce energy to a certain PE amount, to which adding additional PE results in CPU cycles being consumed for barrier spin locks. Although the CPU cycles are wasted, Dynamic Frequency and Voltage Scaling (DVFS) or power gating may help correct this trend without increasing runtime [16].

With automatic parallelization through means of OpenMP or OSCAR, energy reduction is observed from the improvement of runtime. A minimum energy reduction of 77% and runtime improvement of 8.41x was observed with OSCAR at 16PE for scalable benchmarks SWIM, BT, and CG. Furthermore,



**Fig. 6.** Energy comparison (lower is better) compared to base sequential execution (1PE). Data is normalized.

**Table 2.** Benchmark execution time and total energy of OSCAR+CacheOpt (Median of 25 runs).

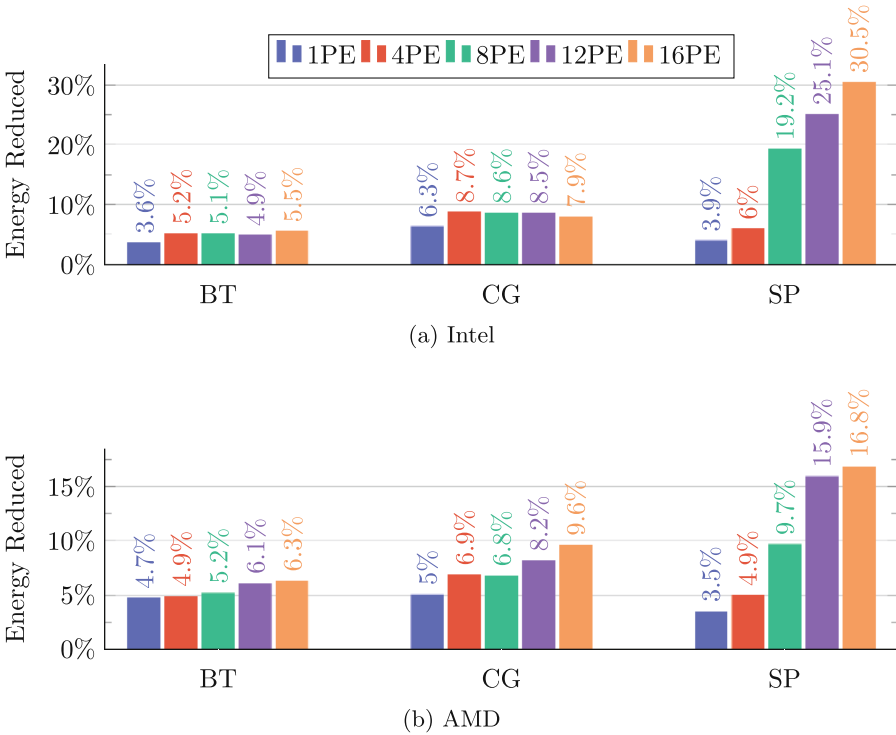
		Time (s)						Energy (J)					
	Benchmark	Sequential	1PE	4PE	8PE	12PE	16PE	Sequential	1PE	4PE	8PE	12PE	16PE
Intel	SWIM	34.85	17.71	5.51	3.89	3.41	3.22	2803	1526	654	557	554	570
	ART	25.44	23.96	10.89	10.85	10.34	10.30	1884	1825	1016	1174	1293	1406
	BT	235.57	227.93	63.41	38.42	30.31	26.31	19424	18729	6931	5139	4667	4442
	CG	118.38	110.71	29.50	17.21	13.27	11.09	9623	9018	3154	2296	2045	1894
	SP	90.67	87.46	38.85	30.57	28.91	27.59	7643	7342	3094	3739	3970	4251
AMD	SWIM	18.43	9.84	3.93	2.26	1.83	1.41	1146	643	296	203	197	176
	ART	18.00	16.17	7.30	6.97	6.70	6.27	1044	939	485	700	834	854
	BT	185.64	177.34	56.82	37.77	27.07	22.07	11313	10783	4155	3057	2541	2327
	CG	66.96	62.18	18.31	9.78	6.89	5.80	4164	3955	1495	992	834	688
	SP	73.69	71.02	34.21	32.74	30.56	30.27	4578	4420	2486	2452	2555	2858



we can see OSCAR with cache optimization providing less energy compared to both OpenMP and OSCAR for equivalent PE used.

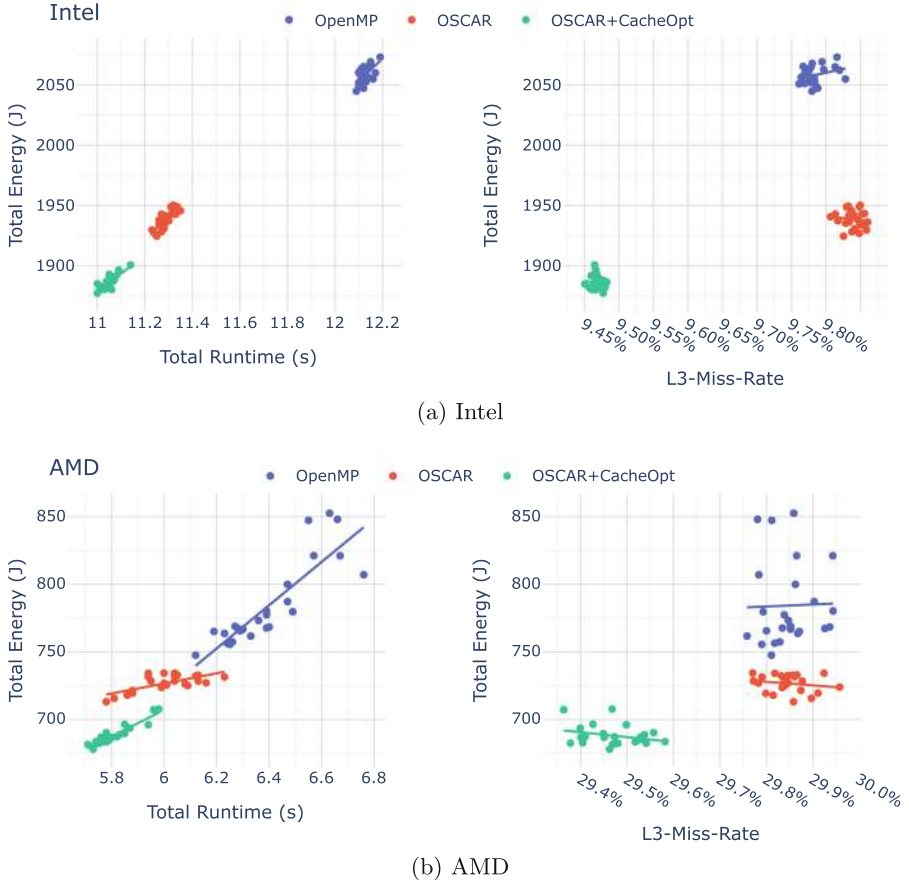
## 5.2 Energy Reduction by OSCAR with Cache Optimizations Compared with OpenMP

When observing energy reduction compared to OpenMP for an equivalent PE or core count, we can see that OSCAR with Cache Optimizations is able to further reduce the total energy consumed through improved runtime. In Fig. 7, we can see a reduction of 5% or more on BT and CG and an increasing reduction on SP for up to 30% on Intel and 15% on AMD. On BT and CG, the 5% or greater reduction is mainly due to cache optimizations from the data-localization-groups being applied to the doall and dosum loops described in Sect. 2.2. As for SP, it reaches a crossover where using over 4PE increases total energy consumption as seen in Table 2. In these cases with saturation, OSCAR is able to provide less overhead with improved runtime and energy compared to OpenMP.



**Fig. 7.** Energy reduction of OSCAR with Cache Optimization versus OpenMP for equivalent PE.

On both Intel and AMD architectures, runtime performance was observed to match or exceed previous OSCAR evaluations on older architectures from Intel and AMD [10]. In addition, direct comparisons to OpenMP and the effects of cache optimizations were not taken prior. Note that for SWIM and ART, only sequential versions of the benchmark were available so they were not included in the analysis and in Fig. 7.



**Fig. 8.** Scatter plots of energy, runtime, and L3 cache performance for NAS CG using 16PE.

### 5.3 Cache Performance and Energy in Parallelized Benchmarks

The benchmarks can be further evaluated to see the trends between cache performance, runtime, and energy. As previously shown, using cache optimization

and data localization techniques provided improved runtime and energy. Looking at CG with 16PE in Fig. 8, we can see distinct distributions between the three different benchmark versions. These scatter plots show the relation of L3 cache performance and the corresponding runtime and energy data points. On both Intel and AMD, OSCAR with cache optimization is providing better results than OpenMP and OSCAR without cache optimizations. For Intel, there is a 4% runtime improvement in addition to a 4% reduction in energy used when OSCAR+CacheOpt is compared with OSCAR. In addition to parallelization providing a 80% energy improvement, cache optimization is driving an additional improvement in runtime, which is driving a proportional improvement in energy. The difference of 0.3% L3 miss rate improvement is driving the reduction in runtime and energy. On AMD we see a similar trend is observed. Notably it achieves a 5% improvement in runtime with a 7% reduction in energy with a 0.5% L3 miss rate improvement. With these values we can see a portion of the energy reduction in Figs. 6 and 7 is coming from cache optimization and data localization. Similar trends were observed for other linear-scaling benchmarks at 16PE.

The difference in Intel and AMD’s network topologies is also noticeable in the data, with AMD realizing greater energy reduction for cache optimization in these conditions at maximum PE. This can be attributed to AMD’s SoC due to the additional latency cost compared to Intel [20], especially for cross CCX communication. On both Intel and AMD with different cache topologies, OSCAR was cache optimizations shown to provide improved L3 performance, providing additional improvements to runtime and energy.

## 6 Conclusion

The OSCAR automatic parallelizing compiler is able to use multigrain parallelism along with cache optimization and data allocation techniques to provide both speedup and energy reduction on x86-64 architectures. Given an input source code file, the OSCAR compile will analyze the control and data dependencies, apply coarse grain task parallelism and optimizations, and generate output code. This generated parallel code by the OSCAR compiler is a set of C or Fortran sequential code for each processor core with data transfers and synchronization management. This code can be taken by standard compilers (e.g. GCC) and compiled for respective systems.

In this paper it was found that the generated code by OSCAR was able to provide automatic parallelization, speedup, and power reduction of scientific applications on Intel Xeon Icelake-SP and AMD EPYC Zen-4 architectures. It showed that by reducing runtime, energy was able to reduced on for benchmarks that scaled with additional PE. Benchmarks showed on average 4% to 9% energy reduction compared to OpenMP for the same amount of PE. On NAS Parallel Benchmark CG for 16PE, OSCAR achieved an increased relative speedup of 10.6x against 1PE on both Intel and AMD in comparison to 9.73x with OpenMP. Furthermore, using cache optimization and data localization techniques yielded

a L3 cache miss rate improvement of 0.2% to 0.5% which resulted in an additional energy reduction of 4% to 7% (42 J to 50 J) from improved runtime. When observing different PE performance for OpenMP versus OSCAR for the NAS Parallel Benchmarks, cache improvements yielded a range of 4% to 9% of energy reduction achievable.

With optimal cache optimization techniques and data localization, cache performance can have a notable impact on energy reduction in addition to runtime improvements. Depending on the architecture's network topology, systems with higher memory latency are able to achieve greater energy reduction. It is shown that using cache optimization and data localization on Intel and AMD multicore systems yielded cache improvements resulting in energy reduction and improving runtime on scientific benchmarks.

## References

1. Real world computing project: Omni openmp compiler project. WebPage. <http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/>. Accessed 07 Mar 2024
2. Running average power limit energy reporting. Tech. rep., Intel (November 2020). <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html>
3. 4th gen amd epyc processor architecture white paper. Tech. rep. (May 2024). <https://www.amd.com/system/files/documents/4th-gen-epyc-processor-architecture-white-paper.pdf>
4. Bailey, D., et al.: The NAS parallel benchmarks. *Int. J. Supercomput. Appl.* **5**(3), 63–73 (1991). <https://doi.org/10.1177/109434209100500306>
5. Hirano, T., et al.: Evaluation of automatic power reduction with OSCAR compiler on intel Haswell and arm cortex-a9 multicores. In: Brodman, J., Tu, P. (eds.) *Languages and Compilers for Parallel Computing*, pp. 239–252. Springer International Publishing, Cham (2015)
6. Dietz, H.G. (ed.): *LCPC 2001*. LNCS, vol. 2624. Springer, Heidelberg (2003). <https://doi.org/10.1007/3-540-35767-X>
7. Kasahara, H., et al.: Oscar parallelizing and power reducing compiler and API for heterogeneous multicores : (invited paper). In: *2021 IEEE/ACM Programming Environments for Heterogeneous Computing (PEHC)*, pp. 10–19 (2021). <https://doi.org/10.1109/PEHC54839.2021.00007>
8. Kimura, K., Mase, M., Mikami, H., Miyamoto, T., Shirako, J., Kasahara, H.: OSCAR API for real-time low-power multicores and its performance on multicores and SMP servers. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) *LCPC 2009*. LNCS, vol. 5898, pp. 188–202. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-13374-9\\_13](https://doi.org/10.1007/978-3-642-13374-9_13)
9. Li, Y., Abousamra, A., Melhem, R., Jones, A.K.: Compiler-assisted data distribution for chip multiprocessors. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. p. 501–512. PACT '10, Association for Computing Machinery, New York, NY, USA (2010). <https://doi.org/10.1145/1854273.1854335>

10. Magnussen, B.M., Kawasumi, T., Mikami, H., Kimura, K., Kasahara, H.: Performance evaluation of OSCAR multi-target automatic parallelizing compiler on Intel, AMD, Arm and RISC-V Multicores. In: Li, X., Chandrasekaran, S. (eds.) LCPC 2021. LNCS, vol. 13181, pp. 50–64. Springer, Cham (2022). [https://doi.org/10.1007/978-3-030-99372-6\\_4](https://doi.org/10.1007/978-3-030-99372-6_4)
11. Mulnix, D., Kumar, A., Ould-ahmed-vall, E.: Intel Xeon processor scalable family technical overview. Tech. rep., Intel (December 2022). <https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html>
12. Palkowski, M., Gruzewski, M.: Time and energy benefits of using automatic optimization compilers for NPDP tasks. *Electronics* **12**(17) (2023). <https://doi.org/10.3390/electronics12173579>, <https://www.mdpi.com/2079-9292/12/17/3579>
13. Papazian, I.E.: New 3rd gen intel® xeon® scalable processor (codename: Ice lake-sp). In: 2020 IEEE Hot Chips 32 Symposium (HCS), pp. 1–22 (2020). <https://doi.org/10.1109/HCS49909.2020.9220434>, [https://hc32.hotchips.org/assets/program/conference/day1/HotChips2020\\_Server\\_Processors\\_Intel\\_Irma\\_ICX-CPU-final3.pdf](https://hc32.hotchips.org/assets/program/conference/day1/HotChips2020_Server_Processors_Intel_Irma_ICX-CPU-final3.pdf)
14. Poolla, C., Saxena, R.: On extending Amdahl’s law to learn computer performance. *Microprocess. Microsyst.* **96**, 104745 (2023)
15. Shirako, J., Oshiyama, N., Wada, Y., Shikano, H., Kimura, K., Kasahara, H.: Compiler control power saving scheme for multi core processors. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) *Languages and Compilers for Parallel Computing*, pp. 362–376. Springer, Berlin Heidelberg, Berlin, Heidelberg (2006)
16. Shrivastava, R., Nandivada, V.K.: Energy-efficient compilation of irregular task-parallel loops. *ACM Trans. Archit. Code Optim.* **14**(4) (nov 2017). <https://doi.org/10.1145/3136063>
17. Talati, N., et al.: Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In: 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 654–667 (2021). <https://doi.org/10.1109/HPCA51647.2021.00061>
18. Troester, K., Bhargava, R.: AMD next generation “zen 4” core and 4th gen amd epyc™ 9004 server CPU. In: 2023 IEEE Hot Chips 35 Symposium (HCS), pp. 1–25 (2023). <https://doi.org/10.1109/HCS59251.2023.10254726>, [https://hc2023.hotchips.org/assets/program/conference/day1/CPU1/HC\\_Zen4\\_Epyc\\_Final\\_20230825%20-%20Embargoed%20until%20Aug%2029%202023.pdf](https://hc2023.hotchips.org/assets/program/conference/day1/CPU1/HC_Zen4_Epyc_Final_20230825%20-%20Embargoed%20until%20Aug%2029%202023.pdf)
19. Umeda, D., Suzuki, T., Mikami, H., Kimura, K., Kasahara, H.: Multigrain parallelization for model-based design applications using the OSCAR compiler. In: Shen, X., Mueller, F., Tuck, J. (eds.) *LCPC 2015*. LNCS, vol. 9519, pp. 125–139. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-29778-1\\_8](https://doi.org/10.1007/978-3-319-29778-1_8)
20. Velten, M., Schöne, R., Ilsche, T., Hackenberg, D.: Memory performance of amd epyc rome and intel cascade lake sp server processors. In: *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, pp. 165–175. ICPE ’22, Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3489525.3511689>, <https://doi.org/10.1145/3489525.3511689>

21. Yoshida, A., Koshizuka, K., Kasahara, H.: Data-localization for Fortran macro-dataflow computation using partial static task assignment. In: Proceedings of the 10th International Conference on Supercomputing, pp. 61–68. ICS '96, Association for Computing Machinery, New York, NY, USA (1996). <https://doi.org/10.1145/237578.237586>
22. Yuki, T., Rajopadhye, S.: Folklore confirmed: Compiling for speed  $\neq$  compiling for energy. In: Cacaval, C., Montesinos, P. (eds.) Languages and Compilers for Parallel Computing, pp. 169–184. Springer International Publishing, Cham (2014)



# Concurrent Collections: An Overview

Kathleen Knobe<sup>1</sup>, Zoran Budimlic<sup>2</sup>, Robert J. Harrison<sup>3</sup>,  
Mohammad Mahdi Javanmard<sup>3</sup>, and Louis-Noël Pouchet<sup>4</sup>✉

<sup>1</sup> Rice University, Houston, USA

<sup>2</sup> Texas A&M University, Bryan, USA

<sup>3</sup> IACS, Stony Brook University, Stony Brook, USA

<sup>4</sup> Colorado State University, Fort Collins, USA

pouchet@colostate.edu

**Keywords:** Concurrent Collections · Dataflow · Distributed Computing

## Introduction

Parallel programming is essential to achieve high-performance, and numerous works combined programming languages, runtime and compilers to help the deployment of effective high-performance applications at scale. Many recent programming models allow a specification of a task graph representing the application to be created by the programmer. For example, the *depends* clause in OpenMP 4.0 allows the programmers to create arbitrary dependences between OpenMP tasks. Habanero Data Driven Tasks [21] and OCR Event Driven Tasks and Events [1] provide similar capabilities as well. While these systems enable the creation of task graphs, they exhibit varying degrees of *separation of concerns*, or decoupling of program correctness from performance.

Concurrent Collections (CnC) is a parallel programming model, with an execution semantics that is influenced by dynamic data flow, stream processing, and tuple spaces [5]. CnC was developed in part to address the need for making parallel programming accessible to non-expert developers. It relies on users to specify explicitly the data and control dependences between tasks, in turn allowing automating the generation [18] of high-performance parallel programs for a variety of targets [7], from distributed computers to GPUs [10] to FPGAs [9]. A CnC program has deterministic semantics: Any implementation of it that follows the dependences specified in the CnC program will produce the same outputs. This deterministic semantics, and the separation of concerns between the domain experts in charge of describing the core application dependences, and the tuning experts mapping this program to a particular hardware are the primary characteristics that differentiate CnC from other parallel programming models.

In this paper, we will discuss the fundamental concepts of CnC, its history, and its achievements in terms of programmability and performance. We will describe several iterations of CnC implementations, including CnC execution models on top of Java [3] and C++ [21,23], as well as several domain-specific

uses of CnC, from general-purpose programming [2], GPGPU-centric [10, 18], exascale-focused [14], to centered on large heterogeneous applications [15].

## Concurrent Collections Programming Model

Concurrent Collections (CnC) is a programming model that evolved out of TStreams [12], a dataflow programming model designed for expressing parallel computations as a network of typed, timestamped data streams.

CnC uses three core constructs: step collections (representing computations), item collections (representing data and holding typed, immutable data items), and control collections (governing execution flow). At runtime, the corresponding dynamic instances—step instances, data items, and control tags—are generated, each belonging to a static collection. Step and item instances are uniquely identified by a collection name and a *tag*, which is typically an integer tuple encoding some useful information about the computation or data, such as position in the iteration space. Data elements are inserted and retrieved using the `put (item, tag)` and blocking `get (tag)` operations, ensuring determinism and freedom from data races due to the single assignment semantics [2]. Step collections may consume from or produce to item collections, or do both. Control collections manage step instantiation via control relationships, where adding a tag to a control collection enables (“prescribes”) step instances in associated step collections.

Chandramowliswaran et al. [7] showed that scientific CnC programs could exhibit performance that is on par or exceeding the state of the art parallel programming systems, which sparked a significant interest in the model.

The deterministic nature of CnC and its explicit handling of data and control dependencies have inspired several extensions and improvements to the original programming model. Grossman et al. [10] proposed extensions to allow creation of GPU tasks, Imam [11] proposed a Python-driven CnC implementation that allowed multi-language step implementations using Babel.

Chatterjee et al. [8] introduced *TunedCnC*, a declarative tuning framework that extends the Concurrent Collections (CnC) model with two additional constructs: *affinity groupings* and *distribution functions*. Affinity groupings allow computations to be organized into groups based on tags, promoting data locality, while distribution functions specify how computations and data are mapped across available compute resources.

A. Sbirlea et al. [18] explored mapping CnC graphs onto heterogeneous platforms to enhance performance and energy efficiency. To support this, they extended the CnC model with *tag functions* and *ranges*. Tag functions enable users to define relationships between step tags and the dynamic instances they create or access, while ranges facilitate efficient bulk operations—such as reading or writing groups of items and prescribing groups of steps.

D. Sbirlea et al. [19] examined the use of the CnC model in streaming applications by introducing Streaming Concurrent Collections (SCnC)—a restricted version of the CnC programming model tailored for streaming execution. SCnC



includes a code generator and a runtime library designed to optimize performance specifically for streaming workloads.

Milaković [15] introduced two novel extensions to the CnC model: *Unified CnC* and *Hierarchical CnC*. The Unified CnC extension streamlines CnC graph specification, enabling users to prescribe steps by simply adding data to a collection. The Hierarchical CnC extension allows users to organize data and computation hierarchically. With this hierarchy, the CnC runtime (or compiler) can group multiple fine-grained steps into a single coarse-grained step, reducing the number of tasks and accesses to concurrent data structures. Additionally, the hierarchy helps with garbage collection by enabling collection at a coarse-grained level, reducing the overhead of tracking references for fine-grained data.

## Runtime Implementations

The original CnC implementation was developed at Intel [20], using C++ as the step language and Intel TBB [16] as the underlying task execution model. Due to the declarative and deterministic nature of CnC, this led to a relatively straightforward runtime extension to allow CnC execution on distributed memory systems [20]. Vasilache et al. [22] discuss trade-offs for event-driven runtimes, on top of which CnC can be implemented.

Budimlić et al. [4] Used Java as the step programming language, and the Habanero Java [6] runtime as the underlying task execution model. Sbirlea et al. [18] extended this implementation with a runtime that supports execution of CPU, GPU, and FPGA steps.

Milaković [15] developed an efficient Hierarchical CnC runtime based on Intel’s CnC C++ implementation, introducing the concept of *micro-runtimes*. A micro-runtime groups multiple fine-grained steps into a higher-level step, communicating with the macro-runtime through high-level `puts` and `gets`. From the macro-runtime’s perspective, each micro-runtime is simply a Hierarchical CnC step. Like the macro-runtime, micro-runtimes have step and item collections, where each item collection corresponds to a coarse-grained item. Micro-runtimes translate fine-grained item accesses into coarse-grained ones. In his implementation, micro-runtimes sequentially execute their fine-grained steps, which, while not a fundamental requirement, reduces parallelism and concurrency overhead and promotes data reuse. Since fine-grained steps within the same micro-runtime typically operate on shared data, sequential execution improves cache efficiency by keeping common data in memory.

## Compiler Support for CnC

The explicit nature of dependencies in CnC, and the explicit description in CnC programs of the data items being read or written by task instances provide a parallel program representation that is amenable to compiler optimization. In particular, the use of tag functions to express the relation between dynamic

instances of tasks and the data they manipulate enables to develop new compiler analyses targeting CnC programs.

Of the numerous works on compiler support for CnC programs, for example Sbirlea et al. [17] introduce a new optimization framework for the Data-Flow Graph Language (DFGL), a programming model based on CnC. This framework uses a “dependence-first” approach to capture program semantics in polyhedral representations, performs legality checks on DFGL programs and enables polyhedral transformations, including automatic loop optimizations and parallel code generation. Performance experiments show that DFGL programs optimized by this framework achieve up to 6.9x speedup over standard OpenMP implementations on multicore processors.

Kong et al. [13] introduce PIPES, an end-to-end programming framework for CnC, which automatically generates Intel CnC C++ runtime code from high-level textual description of a CnC graph. In particular, it performs polyhedral analysis of the input graph, enabling to restructure it automatically for example by implementing polyhedral tiling to coarsen fine-grain tasks into larger-grain ones automatically, reducing runtime overhead and improving performance. PIPES automatically generates tuners for CnC, from the analysis of the input graph, to improve further performance. It demonstrated the ability to outperform reference high-performance implementations such as ScalaPack using tuned CnC implementations [13].

## Conclusions

The declarative and deterministic nature of CnC, coupled with its explicit data and computation dependencies, has driven significant research in parallel and distributed computing, leading to numerous extensions of the model and its execution. The ability to express maximum implicit parallelism offers substantial potential for further exploration, while the explicit dependencies provide opportunities for advanced compiler optimizations. As a result, CnC continues to be a compelling platform for research in compiler design, runtime systems, high-performance computing, and programming languages. Its ongoing relevance underscores its value as a versatile and powerful model for addressing complex computational challenges.

## References

1. Open Community Runtime. <https://xstackwiki.modelado.org/OCR>
2. Budimlić, Z., Burke, M., et al.: Concurrent Collections. Scientific Programming (2010)
3. Budimlić, Z., Chandramowlishwaran, A., Knobe, K., Lowney, G., Sarkar, V., Treggiari, L.: Declarative aspects of memory management in the concurrent collections parallel programming model. In: Workshop on Declarative Aspects of Multicore Programming (DAMP) (2009)

4. Budimlić, Z., Chandramowlishwaran, A., Knobe, K., Lowney, G., Sarkar, V., Treggiari, L.: Multi-core Implementations of the Concurrent Collections Programming Model. In: International Conference on Principles and Practice of Programming in Java (CPC) (2009)
5. Burke, M.G., Knobe, K., Newton, R., Sarkar, V.: Concurrent Collections Programming Model. *Encyclopedia of Parallel Computing* (2011)
6. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-Java: the New Adventures of Old X10. In: International Conference on the Principles and Practice of Programming in Java (PPPJ) (2011)
7. Chandramowlishwaran, A., Knobe, K., Vuduc, R.: Performance evaluation of concurrent collections on high-performance multicore computing systems. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS) (2010)
8. Chatterjee, S., Vrvilo, N., Budimlić, Z., Knobe, K., Sarkar, V.: Declarative tuning for locality in parallel programs. In: International Conference on Parallel Processing (ICPP) (2016)
9. Cong, J., Gururaj, K., Zhang, P., Zou, Y.: Task-level data model for hardware synthesis based on concurrent collections. *J. Electr. Comput. Eng.* (2012)
10. Grossman, M., Sbirlea, A., Budimlić, Z., Sarkar, V.: CnC-CUDA: declarative programming for GPUs. In: International Workshop on Languages and Compilers for Parallel Computing (LCPC) (2010)
11. Imam, S., Sarkar, V.: CnC-Python: multicore programming with high productivity. In: *USENIX Workshop on Hot Topics in Parallelism (HotPar)* (2012)
12. Knobe, K., Offner, C.: Tstreams: how to write a parallel program. Hewlett Packard Technical Report (2004)
13. Kong, M., Pouchet, L.N., Sadayappan, P., Sarkar, V.: Pipes: a language and compiler for task-based programming on distributed-memory clusters. In: *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 456–467. IEEE (2016)
14. Madsen, T.G., et al.: The open community runtime: a runtime system for extreme scale computing. In: *IEEE High Performance Extreme Computing Conference (HPEC)* (2016)
15. Milaković, S.: Compiler and Runtime Optimization of Computational Kernels for Irregular Applications. Ph.D. thesis, Rice University (2023)
16. Reinders, J.: *Intel Threading Building Blocks*. O'Reilly Media (2007)
17. Sbirlea, A., Shirako, J., Pouchet, L.N., Sarkar, V.: Polyhedral optimizations for a data-flow graph language. In: *Languages and Compilers for Parallel Computing* (2016)
18. Sbirlea, A., Zou, Y., Budimlić, Z., Cong, J., Sarkar, V.: Mapping a Data-Flow Programming Model onto Heterogeneous Platforms. *ACM SIGPLAN Notices* (2012)
19. Sbirlea, D., Shirako, J., Newton, R., Sarkar, V.: SCnC: efficient unification of streaming with dynamic task parallelism. In: *Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM)* (2011)
20. Schlimbach, F., Brodman, J.C., Knobe, K.: Concurrent collections on distributed memory theory put into practice. In: *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)* (2013)
21. Taşirlar, S., Sarkar, V.: Data-driven tasks and their implementation. In: *International Conference on Parallel Processing (ICPP)* (2011)
22. Vasilache, N., et al.: A tale of three runtimes. *arXiv preprint [arXiv:1409.1914](https://arxiv.org/abs/1409.1914)* (2014)
23. Vrvilo, N.: Enhanced Data and Task Abstractions for Extreme-scale Runtime Systems. Ph.D. thesis, Rice University (2017). <https://habanero.rice.edu/vrvilo-phd>



# Hidden Assumptions in Static Verification of Data-race Free GPU Programs

Tiago Cogumbreiro<sup>1</sup>(✉)  and Julien Lange<sup>2</sup> 

<sup>1</sup> University of Massachusetts Boston, Boston, USA  
tiago.cogumbreiro@umb.edu

<sup>2</sup> Royal Holloway, University of London, London, UK  
julien.lange@rhul.ac.uk

**Abstract.** GPUs are massively parallel devices that promise a great return of investment at a cost: GPUs are notably difficult to get right. We discuss a static analysis tool for GPU programs, called **Faial**, that can detect data-races and data-race freedom. We studied a dataset of 191 data-race free programs and found that 98% needs specific thread configuration to be analyzable, and that 27% needs user-provided assertions to be analyzable. We also report that **Faial** was able to find data-races in at least 92% of the kernels with missing assumptions.

## 1 Introduction

For the last 20 years, Vivek Sarkar has been studying the problem of analyzing a data-races in parallel programs both statically [6, 29–32] and dynamically [8, 12, 17, 27, 28, 33]. A data-race is a bug characterized by two unsynchronized memory accesses targeting the same location by different threads, where at least one access is a store. This paper focuses on data-races that arise in the context of GPU programs (also called kernels). GPUs have been widely successful in propelling the scientific advancement of a series of research fields, such as Artificial Intelligence, Machine Learning, molecular modeling, systems biology, and medical imaging.

*Data-race detection* is a program verification technique that proves the existence of a data-race in a possible run of a program. The most common approach to detect data-races is with dynamic analysis, by monitoring the execution of the program to find data-races. Many dynamic analysis techniques have been proposed [13, 16, 18, 23, 25, 35, 36]. However, since the runtime overhead of dynamic analysis is of 10× up to 1,000× and require the program’s input, dynamic analysis is more applicable to testing. Symbolic execution and model checking can be used to detect data-races without needing the program’s input, however the overheads can be even higher due to the state explosion problem [21, 22, 26].

Data-races can also be detected statically, thus sidestepping the runtime overheads. *Data-race freedom (DRF) detectors* for GPU programs [4, 5, 9, 10, 19, 20] can guarantee that a program is free from data-races, in the analysis of GPU programs. When a DRF detector is unable to prove that a program is DRF,

```

__global__ void saxpy(int n, float a, float *x, float *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

```

**Fig. 1.** A simple GPU program.

it generates an alarm that represents a *potential* data-race, *i.e.*, alarms may be spurious. Such techniques can be used to find data-races, by manually validating the alarms. Yet, since these tools are *unable* to guarantee that the alarms are true, we do not consider this family of tools to be data-race detectors. To the best of our knowledge, and excluding symbolic execution and model checking approaches, Yuki *et al.* were among the first to introduce a static race detector [34], for X10 programs. Chatarasi *et al.* proposed the first static race detector for OpenMP and openACC [7], Gorogiannis *et al.* introduce the first static race detector for multithreaded programs [14], and Liew *et al.* introduce the first static race detector for GPU programs [24].

In this paper we evaluate Faial [24], a data-race and DRF detector for GPU programs. We investigate how different features of the analysis affect DRF detection in a dataset that only contains data-race free kernels. We also investigate whether the tool can report data-races when it lacks information to prove DRF.

The outline of this paper is as follows. Section 2 gives some background by introducing GPU programming as well as discussing implicit assumptions that are needed to prove DRF. Section 3 introduces and tests our research questions. Finally, Sect. 4 summarizes our findings.

## 2 Background

In this section, we give a quick background on GPU programming. We then show that even trivial GPU programs include multiple implicit assumptions.

### 2.1 GPU Programming

SAXPY (Single-Precision  $A \cdot X$  Plus  $Y$ ) is a classic example that showcases the kind of numeric applications that run on GPU devices. Given two vectors of floating points  $X$  and  $Y$  and a scalar  $A$  the program updates vector  $Y$  such that  $Y[i]$  stores the result of  $A \cdot X[i] + Y[i]$  for each element  $i$ . A SAXPY operation can be implemented as a GPU program in Fig. 1. In this paper we use the CUDA Application Programming Interface (API); the same concepts apply to other GPU programming models. A GPU executes function `saxpy` for a certain number of threads arranged in groups. Each group of threads is called a *block*. The threads of a block are logically arranged in a 3-D space, each thread is uniquely identified by a 3D point accessible with variable `threadIdx`. The set of all blocks is also logically arranged in a 3-D space, each block is uniquely identified with a 3-D point accessible with variable `blockIdx`. The number of threads

per block, *i.e.*, the block layout, is accessible in variable `blockDim`. The number of blocks in the system are given by variable `gridDim`. A GPU program runs a copy of function `saxpy` per thread in parallel, instantiating variables `threadIdx` and `blockIdx` for each thread. Variable  $i$  represents a unique thread across all groups, since it projects the  $x$ -component of variables `blockIdx` with `threadIdx` onto a linear space. A *thread configuration* is defined as the number of blocks and the number of threads per block.

When a kernel is data-race free only under certain assumptions we call that kernel *partially data-race free*. For instance, the example in Fig. 1, taken from a tutorial on CUDA programming [15], is partially data-race free. Next, we show two assumptions that render Fig. 1 partially data-race free: thread configurations, and grid-level synchronization.

**Ranging Over All Thread Configurations.** The statement that variable  $i$  is unique thread across all groups only holds when there is only one dimension in the  $y$  and  $z$  axis. Hence, a data-race exists between thread `threadIdx = {x = 0, y = 1, z = 1}` and `threadIdx = {x = 0, y = 0, z = 0}` both from block `blockIdx = {x = 0, y = 0, z = 0}` for a block `blockDim = {x = 1, y = 2, z = 2}`, *i.e.*,  $2 \times 2$  threads in the  $y$ - $z$  axis. The data-race occurs because the projection in variable `i` assumes that all threads are arranged in dimension  $x$ , yet a data-race can occur if there are threads in dimensions  $y$  and  $z$ . We can add an assertion to make this fact explicit:

```
__assume(blockDim.y == 1 && blockDim.z == 1);
```

**Grid-Level Synchronization.** The distinction between block-level and grid-level analysis is important to the analysis, because different kinds of memory can be shared at different levels, and also synchronization mechanisms are available at different levels. If we consider data-races across different blocks, then another data-race is possible. For instance, between thread `threadIdx = {x = 0, y = 0, z = 0}` of block `blockIdx = {x = 0, y = 0, z = 0}` and thread `threadIdx = {x = 0, y = 0, z = 0}` of block `blockIdx = {x = 0, y = 1, z = 1}`. We can add an assertion to make the data-race freedom assumption explicit: `__assume(gridDim.y == 1 && gridDim.z == 1);`

We list the kernel with both user-provided assertions that are needed to prove data-race freedom in Fig. 2.

### 3 Evaluation

Faial is the only tool capable of data-race and data-race-freedom detection. Given a data-set of kernels identified as data-race free, we pose two research questions:

**RQ1:** *Which analysis features affect partial data-race freedom?* We select different features and measure how many kernels cannot be analyzed to understand the impact each feature has in this dataset.

```

__global__ void saxpy(int n, float a, float *x, float *y) {
  __assume(blockDim.y == 1 && blockDim.z == 1);
  __assume(gridDim.y == 1 && gridDim.z == 1);
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

```

**Fig. 2.** A simple example with user-provided assumptions.

**RQ2:** *Can static data-race detection help with missing assumptions?* In the context of this experiment, racy kernel indicate missing assumptions. Since Faial is not guaranteed to find every possible data-race, we want to test if we can use Faial to detect data-races in kernels with missing assumptions.

Both research questions consider 5 experiments. Each experiment runs Faial on the same 191 kernels with different analysis settings. The tool can report that the kernel is data-race free, racy, or timeout.

**Data Selection.** The dataset we use is taken from a benchmark suite of GPU kernels [2]. The dataset is well studied as it has been used in multiple published papers on static analysis of data-races in GPU kernels [2,9,10,24]. The dataset consists of CUDA kernels from 4 benchmark suites: NVIDIA GPU Computing SDK v2.0 (8 kernels), NVIDIA GPU Computing SDK v5.0 (165 kernels), Microsoft C++ AMP Sample Projects (20 kernels), gpgpu-sim benchmarks [1] (33 kernels). Every kernel is annotated with verification-specific conditions: a thread configuration and optionally user-provided assumptions.

We pick 191 kernels that are deemed data-race free by Faial. Some kernels include user-provided assertions created by the authors of the dataset [2]. Most commonly, the user-provided assertions are stating that a certain variable has fixed value, for instance that the height of a matrix is of some arbitrary size, say 512. Importantly, the user-provided assertions are not constraining the thread configurations, *e.g.*, like we did in Fig. 2.

### 3.1 RQ1: Which Analysis Features Affect Partial Data-Race Freedom?

Table 1 lists the 5 experiments that were performed according to the output of the analysis, data-race free, racy, or timeout.

**Discussion.** Experiment 1 is our baseline, since all kernels can be checked as data-race free, yet note that grid-level analysis is not performed. In experiment 2, we enable grid-level analysis and note that Faial is unable to analyze 5 kernels. Faial delegates a step of data-race freedom analysis (index equality) to the Z3 [11] Satisfiability Modulo Theories (SMT) solver. We are able to verify *all* kernels by setting the SMT solver’s theory to AUFLIA, which assumes closed formulas of

**Table 1.** Column **Id** holds an identifier of the experiment. Column **Block** states whether block-level synchronization is checked. Column **Grid** states whether grid-level synchronization is checked. Column **Assert** states whether user-provided assertions are used. Column **FixThr** states whether a fixed thread configuration is used. Column **DRF** counts the kernels identified as data-race free. Column **Racy** counts the kernels identified as racy. Column **Unk** counts the kernels where the analysis is unable to detect data-race freedom nor data-races. Column **T/O** counts the kernels where the analysis timed out. We include the percentage of kernels over the total number of kernels under analysis.

Id	Block	Grid	Assert	FixThr	DRF	(%)	Racy	(%)	Unk	(%)	T/O	(%)
1	Y	N	Y	Y	191	100%	0	0%	0	0%	0	0%
2	Y	<u>Y</u>	Y	Y	186	97%	0	0%	0	0%	5	3%
3	Y	N	<u>N</u>	Y	139	73%	49	26%	3	2%	0	0%
4	Y	N	Y	<u>N</u>	3	2%	173	91%	15	8%	0	0%
5	Y	<u>Y</u>	<u>N</u>	<u>N</u>	2	1%	173	91%	13	7%	3	2%

linear integer arithmetic extended with free sort and function symbols. In experiment 3, we disable user-provided assertions. Only 26% of the kernels require user-provided annotations to prove data-race freedom.

In experiment 4, we range over all possible thread configurations, rather than using a specific thread configuration. Almost every kernel under analysis (98%) expects a specific thread configuration. Faial would be able to analyze many more kernels fully automatically if it could extract the thread configuration present in the kernel launching codes. Bardsley *et al.* have explored a dynamic analysis technique that extracts the runtime parameters of kernel launches [3].

In experiment 5, we enable grid-level analysis, disable user-provided assertions, and range over all possible thread configurations. We find that there are only 2 kernels that are *fully* data-race free, regardless of the thread configuration and without requiring any user assertions. In one kernel, the only memory accesses are atomics that do not introduce data-races. Atomics are supported by Faial. In the other kernel, the only write access is a benign data-race ignored by Faial. Benign data-races occur when both threads write the same value. Benign data-races are not considered errors. Faial can flag benign data-races as errors if the user chooses.

### 3.2 RQ2: Can Data-Race Detection Help with Missing Assumptions?

In this research question we examine kernels that are *not* considered data-race free by Faial, so either racy, unknown, or have a timeout. We assess whether our tool can detect data-races when there are missing assumptions, *e.g.*, absent thread configuration.

**Discussion.** The results in Table 2 show that the vast majority of kernels (91%) with missing assumptions can be detected by Faial. In our experience, having



**Table 2.** Column **Id** holds an identifier of the experiment. Column **Block** states whether block-level synchronization is checked. Column **Grid** states whether grid-level synchronization is checked. Column **Assert** states whether user-provided assertions are used. Column **FixThr** states whether a fixed thread configuration is used. Column **Racy/Non-DRF** gives the proportion of number of kernels with data-races detected versus the total number of kernels that are not data-race free. Column **%** gives the percentage of Racy/Non-DRF.

Id	Block	Grid	Assert	FixThr	Racy/Non-DRF	%
3	Y	N	<u>N</u>	Y	49/52	94%
4	Y	N	Y	<u>N</u>	173/188	92%
5	Y	<u>Y</u>	<u>N</u>	<u>N</u>	173/189	92%

a static data-race detector has been quite effective to figuring out the correct analysis settings. In contrast, when relying on the alarms of a data-race-freedom detector, there is always uncertainty whether there is an actual data-race or a spurious one.

### 3.3 Bugs Found

In the course of writing this paper, we discovered bugs in the dataset and in **Faial**. We added assumptions and changed the thread configurations of 4 kernels, since these triggered data-races when grid-level analysis was enabled. In 3 kernels we had to reduce the level of parallelism, by decreasing the number of thread blocks. In 1 kernels, we added a user-provided assumption, a constraint of a template parameter that was mentioned as a source comment, yet absent. We excluded 6 kernels from our evaluation that were being considered fully data-race free by **Faial**, although they are not. Two C++ features are currently unsupported by **Faial**: array addresses being incremented in a loop<sup>1</sup> (affected 3 kernels), and references as function parameters<sup>2</sup> (affected 3 kernels). Since it is quite rare for a kernel to be fully data-race free, experiment 5 proved as an effective sanity check to exercise the correctness of **Faial**.

## 4 Conclusion

In this paper we measured the effect of multiple analysis features when detecting data-race freedom statically, in a dataset of 191 data-race free kernels. We found that 98% of the kernels needed a specific thread configuration to be analyzable and that only 27% of the kernels needed user-provided assertions. These results suggest that to enable a fully automatic static analysis, these tools need to be able to infer valid thread configurations. We also showed that the static

<sup>1</sup> <https://gitlab.com/umb-svl/faial/-/issues/117>.

<sup>2</sup> <https://gitlab.com/umb-svl/faial/-/issues/113>.

race detection of **Faial** was able to find data-races in at least 92% of the kernels studied. The static race detector also helped us identify incorrect thread configurations and missing user-provided assumptions in 4 kernels. Finally, we identified two areas of improvement for **Faial**: 6 kernels were excluded from the evaluation due to limitations of the tool (arrays being updated in loops and references as function parameters), and setting **Faial**'s default SMT theory to **AUFLIA** fixed 5 timeouts.

**Acknowledgments.** This material is based upon work supported by the National Science Foundation under Grant No. 2204986. We thank Francis Alcos, Gregory Blike, Ayden Diel, Samyak Gangwal, Austin Guiney, Ramsey Harrison, Paul Maynard, Udaya Sathiyamoorth, and Hannah Zicarelli for their contributions to **Faial**.

## References

1. Bakhoda, A., Yuan, G.L., Fung, W.W.L., Wong, H., Aamodt, T.M.: Analyzing CUDA workloads using a detailed GPU simulator. In: Proceedings of ISPASS, pp. 163–174. IEEE, Piscataway (2009). <https://doi.org/10.1109/ISPASS.2009.4919648>
2. Bardsley, E., et al.: Engineering a static verification tool for GPU kernels. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 226–242. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-08867-9\\_15](https://doi.org/10.1007/978-3-319-08867-9_15)
3. Bardsley, E., Donaldson, A.F., Wickerson, J.: KernelInterceptor: automating GPU kernel verification by intercepting kernels and their parameters. In: Proceedings of IWOCCL, pp. 1–5. ACM, New York (2014). <https://doi.org/10.1145/2664666.2664673>
4. Betts, A., et al.: The design and implementation of a verification technique for GPU kernels. Trans. Program. Lang. Syst. **37**(3), 1–49 (2015). <https://doi.org/10.1145/2743017>
5. Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: Proceedings of OOPSLA, pp. 113–132. ACM, New York (2012). <https://doi.org/10.1145/2384616.2384625>
6. Chatarasi, P., Shirako, J., Kong, M., Sarkar, V.: An extended polyhedral model for SPMD programs and its use in static data race detection. In: Proceedings of LCPC. LNCS, vol. 10136, pp. 106–120. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-52709-3\\_10](https://doi.org/10.1007/978-3-319-52709-3_10)
7. Chatarasi, P., Shirako, J., Kong, M., Sarkar, V.: An extended polyhedral model for SPMD programs and its use in static data race detection. In: Proceedings of LCPC'16, pp. 106–120. Springer, Heidelberg (2017). [https://doi.org/10.1007/978-3-319-52709-3\\_10](https://doi.org/10.1007/978-3-319-52709-3_10)
8. Choi, J., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridharan, M.: Efficient and precise datarace detection for multithreaded object-oriented programs. In: Proceedings of PLDI, pp. 258–269. ACM (2002). <https://doi.org/10.1145/512529.512560>
9. Cogumbreiro, T., Lange, J., Liew Zhen Rong, D., Zicarelli, H.: Memory access protocols: Certified data-race freedom for GPU kernels. FMSD (2023). <https://doi.org/10.1007/s10703-023-00415-0>

10. Cogumbreiro, T., Lange, J., Rong, D.L.Z., Zicarelli, H.: Checking data-race freedom of GPU kernels, compositionally. In: Proceedings of CAV. LNCS, vol. 12759, pp. 403–426. ACM, New York (2021). [https://doi.org/10.1007/978-3-030-81685-8\\_19](https://doi.org/10.1007/978-3-030-81685-8_19)
11. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of TACAS, pp. 337–340. Springer, Heidelberg (2008)
12. Dimitrov, D.K., Vechev, M.T., Sarkar, V.: Race detection in two dimensions. *ACM Trans. Parallel Comput.* **4**(4), 1–22 (2018). <https://doi.org/10.1145/3264618>
13. Eizenberg, A., Peng, Y., Pigli, T., Mansky, W., Devietti, J.: BARRACUDA: binary-level analysis of runtime RAcEs in CUDA programs. In: Proceedings of PLDI, pp. 126–140. ACM, New York (2017). <https://doi.org/10.1145/3062341.3062342>
14. Gorogiannis, N., O’Hearn, P.W., Sergey, I.: A true positives theorem for a static race detector. *Proc. ACM Program. Lang.* **3**(POPL), 1–29 (2019). <https://doi.org/10.1145/3290370>
15. Harris, M.: An easy introduction to CUDA C and C++ (2012). <https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/>. Accessed 9 July 2024
16. Holey, A., Mekkat, V., Zhai, A.: HAccRG: hardware-accelerated data race detection in GPUs. In: Proceedings of ICPP, pp. 60–69. IEEE, Piscataway (2013). <https://doi.org/10.1109/ICPP.2013.15>
17. Jin, F., Yu, L., Cogumbreiro, T., Shirako, J., Sarkar, V.: Dynamic determinacy race detection for task-parallel programs with promises. In: Proceedings of ECOOP. LIPIcs, vol. 263, pp. 1–30. Schloss Dagstuhl, Dagstuhl (2023). <https://doi.org/10.4230/LIPIcs.ECOOP.2023.13>
18. Kamath, A.K., George, A.A., Basu, A.: ScoRD: a scoped race detector for GPUs. In: Proceedings of ISCA, pp. 1036–1049. IEEE, Piscataway (2020). <https://doi.org/10.1109/ISCA45697.2020.00088>
19. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: Proceedings of FSE, pp. 187–196. ACM, New York (2010). <https://doi.org/10.1145/1882291.1882320>
20. Li, G., Gopalakrishnan, G.: Parameterized verification of GPU kernel programs. In: Proceedings of IPDPSW, pp. 2450–2459. IEEE, Piscataway (2012). <https://doi.org/10.1109/IPDPSW.2012.302>
21. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: Proceedings of PPoPP, vol. 47, pp. 215–224. ACM, New York (2012). <https://doi.org/10.1145/2370036.2145844>
22. Li, P., Li, G., Gopalakrishnan, G.: Practical symbolic race checking of GPU programs. In: Proceedings of SC, pp. 179–190. IEEE, Piscataway (2014). <https://doi.org/10.1109/SC.2014.20>
23. Li, P., et al.: LD: low-overhead GPU race detection without access monitoring. *Trans. Arch. Code Optim.* **14**(1), 1–25 (2017). <https://doi.org/10.1145/3046678>
24. Liew, D., Cogumbreiro, T., Lange, J.: Sound and partially-complete static analysis of data-races in GPU programs. *Proc. ACM Program. Lang.* **8**(OOPSLA2) (2024). <https://doi.org/10.1145/3689797>
25. Peng, Y., Grover, V., Devietti, J.: CURD: a dynamic CUDA race detector. In: Proceedings of PLDI, pp. 390–403. ACM, New York (2018). <https://doi.org/10.1145/3192366.3192368>
26. Pereira, P., et al.: Verifying CUDA programs using SMT-based context-bounded model checking. In: Proceedings of SAC, pp. 1648–1653. ACM, New York (2016). <https://doi.org/10.1145/2851613.2851830>

27. Raman, R., Zhao, J., Sarkar, V., Vechev, M.T., Yahav, E.: Efficient data race detection for async-finish parallelism. *Formal Methods Syst. Des.* **41**(3), 321–347 (2012). <https://doi.org/10.1007/S10703-012-0143-7>
28. Raman, R., Zhao, J., Sarkar, V., Vechev, M.T., Yahav, E.: Scalable and precise dynamic datarace detection for structured parallelism. In: *Proceedings of PLDI*, pp. 531–542. ACM (2012). <https://doi.org/10.1145/2254064.2254127>
29. Surendran, R., Raman, R., Chaudhuri, S., Mellor-Crummey, J.M., Sarkar, V.: Test-driven repair of data races in structured parallel programs. In: *Proceedings of PLDI*, pp. 15–25. ACM (2014). <https://doi.org/10.1145/2594291.2594335>
30. Westbrook, E.M., Zhao, J., Budimlic, Z., Sarkar, V.: Practical permissions for race-free parallelism. In: *Proceedings of ECOOP. LNCS*, vol. 7313, pp. 614–639. Springer, Cham (2012). [https://doi.org/10.1007/978-3-642-31057-7\\_27](https://doi.org/10.1007/978-3-642-31057-7_27)
31. Ye, F., Schordan, M., Liao, C., Lin, P., Karlin, I., Sarkar, V.: Using polyhedral analysis to verify OpenMP applications are data race free. In: Laguna, I., Rubio-González, C. (eds.) *Proceedings of CORRECTNESS@SC*, pp. 42–50. IEEE (2018). <https://doi.org/10.1109/CORRECTNESS.2018.00010>
32. Yu, L., Jin, F., Protze, J., Sarkar, V.: Leveraging the dynamic program structure tree to detect data races in OpenMP programs. In: *Proceedings of Correctness@SC*, pp. 54–62. IEEE (2022). <https://doi.org/10.1109/CORRECTNESS56720.2022.00012>
33. Yu, L., Sarkar, V.: GT-Race: Graph traversal based data race detection for asynchronous many-task parallelism. In: Aldinucci, M., Padovani, L., Torquati, M. (eds.) *Proceedings of Euro-Par. LNCS*, vol. 11014, pp. 59–73. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-96983-1\\_5](https://doi.org/10.1007/978-3-319-96983-1_5)
34. Yuki, T., Feautrier, P., Rajopadhye, S., Saraswat, V.: Array dataflow analysis for polyhedral X10 programs. In: *Proceedings of PPOPP*, pp. 23–34. ACM, New York (2013). <https://doi.org/10.1145/2442516.2442520>
35. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: GRace: a low-overhead mechanism for detecting data races in GPU programs. In: *Proceedings of PPOPP*, pp. 135–146. ACM, New York (2011). <https://doi.org/10.1145/1941553.1941574>
36. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: GMRace: detecting data races in GPU programs via a low-overhead scheme. *Trans. Parallel Distrib. Syst.* **25**(1), 104–115 (2014). <https://doi.org/10.1109/TPDS.2013.44>



# Intrepydd: Toward Performance, Productivity, and Portability for Massive Heterogeneous Parallelism

Jun Shirako<sup>(✉)</sup>, Tong Zhou, and Akihiro Hayashi

Georgia Institute of Technology, Atlanta, GA 30332, USA  
{shirako,tz,ahayashi}@gatech.edu

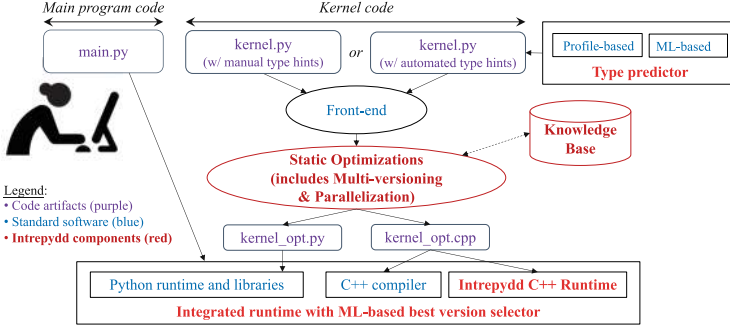
**Abstract.** This paper introduces our ongoing work on the automatic ahead-of-time (AOT) parallelization of Python programs on recent and future hardware systems with massive parallelism and heterogeneity. Our approach is driven by the combination of ML-based type prediction and multi-versioned code generation that guarantees the correctness of our type-specific code optimizations in all cases. While Python is a dynamically-typed language, recent research demonstrated it is highly possible to predict what data types are likely to occur at runtime, by ML-based static prediction and/or runtime type profiling in numerical computation kernels. Given code fragments with predicted data type information, our optimization engine performs automatic parallelization and sophisticated high-level code optimizations for the target system, such as shared/distributed heterogeneous hardware platforms. Our approach introduces novel extensions to the polyhedral compilation to integrate loop and data layout transformations as well as automated selection of CPU vs. GPU code variants. Our preliminary empirical evaluation shows significant performance improvements relative to sequential Python in both single-node and multi-node experiments.

**Keywords:** Parallelizing compilers · Python language · Type prediction · Parallel computing · Heterogeneous computing · Distributed computing

## 1 Introduction

Major simultaneous disruptions are currently underway in both hardware and software. In hardware, massive parallelism and extreme heterogeneity have become critical to sustaining cost and performance improvements after Moore's Law, but pose productivity and portability challenges for developers. In software, the rise of large-scale data science and AI applications is being driven by domain scientists from diverse backgrounds who demand the programmability that they have come to expect from high-level languages like Python. We propose to enable automatic parallelization of sequential Python programs for

recent and future hardware systems with massive parallelism and extreme heterogeneity. We believe that a smart compilation framework that can transform sequential code written in a high-productivity language into an efficient implementation on multicore architectures is highly desirable. The availability of such a framework will help bridge a major productivity gap for domain experts, and reduce the barrier to application enablement on multicore platforms.



**Fig. 1.** Overall Design of Intrepydd system

In this paper, we make a case for new advances to enable productivity and programmability of future multicore platforms for domain scientists. The goal of our framework, Intrepydd, is fully automatic parallelization of standard Python programs, aiming to deliver the benefits of heterogeneous combinations of multicore CPUs, GPU/FPGA accelerators, and future hardware platforms to domain scientists without requiring them to undergo any new training. While Python is a dynamically-typed language, we believe that it is highly possible to predict what data types are likely to occur at runtime, by static prediction based on machine learning [4, 7, 14] and/or runtime type profiling [2]. Our approach includes: 1) multiple candidate type prediction for function parameters and return values; and 2) program multi-versioning for specialized code generation to different candidate data types. After type specialization, we propose to explore a novel approach to automatic ahead-of-time (AOT) parallelization and optimization, which includes: 3) polyhedral-based abstraction and optimizations to fully utilize CPU and GPU parallelism; 4) hybrid Python/C++ code generation that combines high-performance Python library implementations and C-based native code generation; and 5) runtime cost-based automatic selection from various optimization variants including: library-based vs. codegen-based, CPU-based vs. GPU-based, and the combination of those implementation variants.

Figure 1 summarizes the overall design of our proposed Intrepydd system. A user-developed code is a combination of *main program code* and *kernel code*, where the former is unchanged while the latter is optimized by Intrepydd via type prediction, multi-versioning, and automatic AOT source-to-source transformations, which benefit from the use of the Intrepydd Knowledge Base to provide

dataflow and type information for many commonly used library functions. Both execute on a standard Python runtime along with standard libraries used by the application.

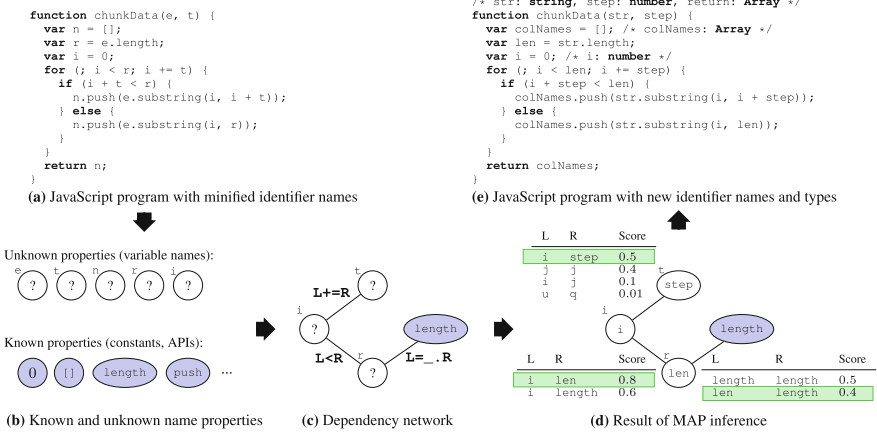


Fig. 2. ML-based type prediction for JavaScript by JSNice [4]

## 2 ML-Based Type Prediction and Runtime Type Profiling

In this work, we integrate two kinds of type prediction approaches, statistical prediction based on machine learning and type sampling based on profiling tools. Figure 2 shows the representative of first approach, JSNice [4], to annotate function parameters and returns with data types. Although JSNice is developed for JavaScript, the same approach is applicable to Python programs by learning with training data from Python applications [14]. There can be many candidates data types for each variable (i.e., function parameter or return value) of interest, especially when integrating both ML-based and profile-based predictions. A big challenge would be to select proper subset of candidate types considering the trade-off between accuracy of type prediction and complexity of multi-versioned code generation. We will address this challenge by the interaction between type predictor and multi-versioner, as discussed in earlier paragraph.

## 3 Polyhedral Optimizations

The polyhedral compilation has provided significant advances in the unification of affine loop transformations combined with powerful code generation techniques [3, 13, 16]. However, despite these strengths in program transformation, the polyhedral frameworks lack support for: 1) dynamic control flow and

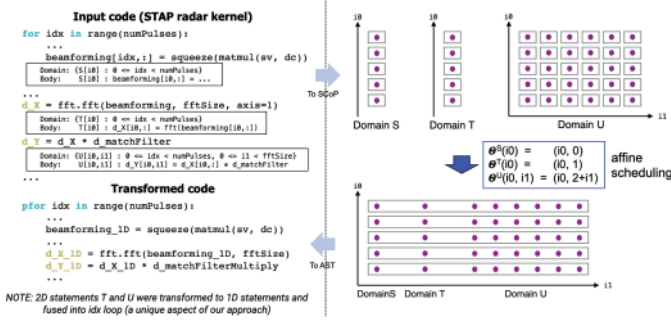


Fig. 3. Polyhedral Optimization

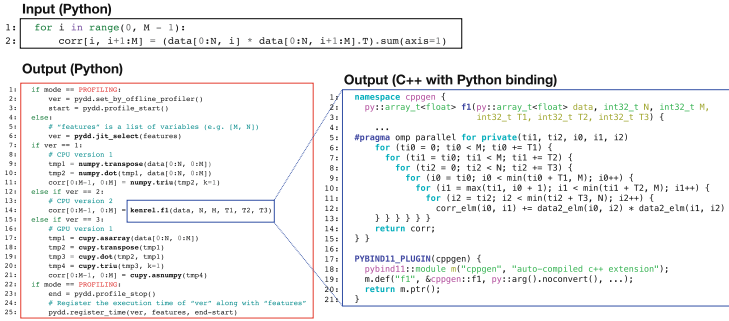


Fig. 4. Hybrid Python/C++ code generation with multi-versioning for Polybench-python correlation example

non-affine access patterns; and 2) library function calls in general. To address the first limitation, we have extended the polyhedral representation of Static Control Parts (SCoPs) to represent unanalyzable expressions as a compound “black-box” statement with approximated input/output relations. To address the second limitation, we took advantage of our library knowledge base to obtain element-wise dataflow relations among function arguments and return values. These unique features enable the co-optimization of both explicit loops and implicit loops from array operators and library calls in a unified optimization framework.

After SCoP extraction and dependence analysis, we can apply any standard polyhedral optimizations to determine the affine scheduling, which composes all the loop and layout transformations in the SCoP representation and is used to generate the transformed Python AST at the code generation step (Fig. 3). In this work, we integrate PolyAST [10] algorithm that implements cache-aware loop transformations and a data layout transformation approach [11, 12] that minimizes the total allocated array sizes while improving spatial data locality. For custom GPU code generation, we develop the Python-to-C++ code generation with OpenMP accelerator model, built on a past work for C-to-CUDA polyhedral optimizer [9].



## 4 Hybrid Python/C++ Code Generation

There are two strategies for polyhedral optimizations in our approach: 1) library mapping that transforms the SCoP representation for select code regions into calls to efficient library functions such as those in NumPy and CuPy; and 2) C++ conversion that enables general loop and layout transformations to maximize parallelism and locality and generates the final output as parallel OpenMP C++ code. While the efficiency of the first strategy was demonstrated [8], we extend the Python-to-C code generation [15] to automatic CPU/GPU parallelization [9,10] for the second strategy. In our approach, an input code region can be optimized with both strategies whenever possible, thereby generating multiple output code versions. Figure 4 illustrates how the input code fragment is optimized with different strategies, version 1: library-based CPU implementation, version 2: C++ codegen-based CPU implementation, and version 3: library-based GPU implementation. The generated C++ code shown on the right side is equipped with the pybind11 [1] APIs with the native OpenMP C++ compiler.

## 5 Preliminary Experimental Results

Figure 5 show the throughput performance of a real-world signal processing application (STAP [5]) OLCF Summit clusters. Given a Python NumPy version as input, prototype Intrepydd compiler automatically parallelized the major computation kernel and mapped it to GPUs via NumPy-to-CuPy conversions. This significantly improves the throughput performance, resulting in comparable single-GPU performance with the manually ported CuPy implementation. The Intrepydd automatically generates API calls to the Ray [6] runtime, which enables scheduling tasks across multiple heterogeneous nodes in a cluster.

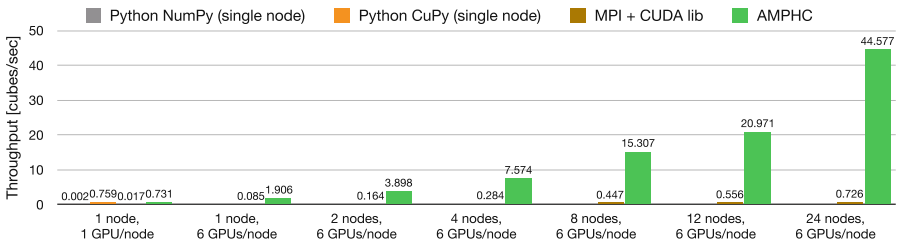


Fig. 5. STAP radar application performance on OLCF Summit supercomputer.

## 6 Conclusions

This paper describes Intrepydd – a programming system designed to deliver the benefits of shared/distributed heterogeneous hardware platforms to domain

scientists who naturally use high-productivity languages like Python. In our approach, the parameters and return values of kernel Python functions are annotated with type hints automatically by the type predictor and their correctness is dynamically checked by the multi-versioned code generation. Based on these type hints, the Intrepydd compiler performs automatic AOT parallelization, including polyhedral-based transformations and CPU/GPU code generation, hybrid Python/OpenMP C++ code generation, runtime cost-based automatic selection. Our empirical evaluations using the STAP radar application for heterogeneous distributed performance show significant performance improvements up to  $20,000\times$  improvement for the STAP radar application, relative to baseline NumPy-based implementations.

## References

1. pybind. <https://pybind11.readthedocs.io/en/stable/> (2015)
2. MonkeyType: Collect run-time types. <https://monkeytype.readthedocs.io/en/latest> (2020). Accessed 25 Aug 2020
3. Bondhugula, U., Acharya, A., Cohen, A.: The pluto+ algorithm: a practical approach for parallelization and locality optimization of affine loop nests. *ACM Trans. Program. Lang. Syst.* **38**(3) (Apr 2016)
4. JS NICE: Statistical Renaming, Type Inference and Deobfuscation. <http://jsnice.org>
5. Melvin, W.L.: Chapter 12: Space-time adaptive processing for radar. Academic Press Library in Signal Processing: Volume 2 Comm. and Radar Signal Proc. (2014)
6. Moritz, P., et al.: Ray: A distributed framework for emerging AI applications. In: *Proceedings of OSDI'18* (2018)
7. Raychev, V., Vechev, M., Krause, A.: Predicting program properties from “big code”. *SIGPLAN Not.* **50**(1), 111–124 (Jan 2015). <https://doi.org/10.1145/2775051.2677009>
8. Shirako, J., Hayashi, A., Paul, S.R., Tumanov, A., Sarkar, V.: Automatic parallelization of python programs for distributed heterogeneous computing. In: *28th International European Conference on Parallel and Distributed Computing (EuroPar)* (2022)
9. Shirako, J., Hayashi, A., Sarkar, V.: Optimized two-level parallelization for GPU accelerators using the polyhedral model. In: *Proceedings of CC 2017* (2017)
10. Shirako, J., Pouchet, L.N., Sarkar, V.: Oil and water can mix: an integration of polyhedral and ast-based transformations. In: *Proceedings of SC'14* (2014)
11. Shirako, J., Sarkar, V.: Integrating data layout transformations with the polyhedral model. In: *Proceedings of IMPACT 2019* (2019)
12. Shirako, J., Sarkar, V.: An affine scheduling framework for integrating data layout and loop transformations. In: *Proceedings of LCPC 2020* (2020)
13. Verdoolaege, S., et al.: Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* **9**(4), 54:1–54:23 (Jan 2013). <http://doi.acm.org/10.1145/2400682.2400713>
14. Ye, F., Zhao, J., Shirako, J., Sarkar, V.: Concrete type inference for code optimization using machine learning with SMT solving (October 2023)
15. Zhou, T., et al.: Intrepydd: performance, productivity and portability for data science application kernels. In: *Proceedings of Onward! '20* (2020)
16. Zinenko, O., et al.: Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling. In: *Proceedings of CC 2018* (2018)



# Enabling User-Level Asynchronous Tasking in the FA-BSP Model Case Study: Distributed Triangle Counting

Akihiro Hayashi<sup>(✉)</sup>, Shubhendra Pal Singhal, Youssef Elmougy,  
and Jiawei Yang

Georgia Institute of Technology, Atlanta, GA, USA  
{ahayashi, ssinghal74, yelmougy3, jyang810}@gatech.edu

**Abstract.** While the FA-BSP model provides significant performance improvements in large-scale graph applications, its single-threaded execution model may limit the performance of certain graph applications. This paper explores the potential benefits of enabling user-level asynchronous tasking with `async` and `finish` in FA-BSP programs using distributed triangle counting. The initial results from a generic HPC cluster show that a version using asynchronous tasking leads to a performance increase of 3% to 32%.

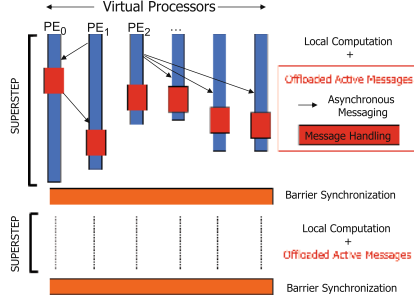
**Keywords:** The FA-BSP Model · Asynchronous Tasking · The Actor Model

## 1 Introduction

The Fine-grained Asynchronous Bulk Synchronous Parallel (FA-BSP) model [11] is an extended version of the BSP model [13] that facilitates fine-grained asynchronous point-to-point messages even during the local computation. As illustrated in Fig. 1, each processing element (PE<sup>1</sup>), performs 1) a local computation (the blue part), 2) asynchronous messaging (the arrows), and 3) message handlers (the red part) in an interleaved fashion. It is important to highlight that the use of the actor model [1] as a user-facing programming model effectively abstracts the execution model as it inherently supports asynchronous messaging and message handling. Such a programming and execution model is perfect for the vertex-centric graph programming model because vertices can efficiently propagate information to their neighbors over the edge via asynchronous messaging. The FA-BSP model typically provides excellent scalability and performance and outperforms state-of-the-art BSP implementations in various large-scale graph applications [4–6, 12].

However, the current FA-BSP model only exploits one-level parallelism, where each PE is single-threaded and performs the interleaved execution. One

<sup>1</sup> In this paper, a PE means an OpenSHMEM PE [2], which is similar to an MPI rank and an OS process.



**Fig. 1.** The Fine-grained Asynchronous Bulk Synchronous Parallel model (FA-BSP).

research question is whether adding an additional level of parallelism is beneficial.

In this paper, we explore the possibility of integrating asynchronous task parallelism with the FA-BSP model with a primary focus on the message-handling part. The key concept is to enable users to create asynchronous tasks within a message handler. These asynchronous tasks can be either 1) synchronized before the message handler ends or 2) allowed to escape from it by relaxing the message processing rule [8].

This paper makes the following contributions:

- Preliminary design and implementation of asynchronous tasking support for an FA-BSP runtime (HClib-Actor).
- Preliminary demonstration showing that using asynchronous tasking in a message handler outperforms the existing single-threaded execution.

## 2 Background

### 2.1 Habanero C/C++ Library (HClib)

The Habanero C/C++ library (HClib) [7] was originally developed to enable an asynchronous many-task (AMT) programming model and its runtime system. It inherits different parallel constructs from the X10 [3] language, such as 1) **finish**, used for bulk task synchronization. It waits on all tasks (including nested tasks) spawned within the scope of the finish, 2) **async**, which is used to create asynchronous tasks, and 3) **async\_at**: a variant of **async**, which is used to spawn asynchronous tasks at a specific location. Note that, unlike X10, HClib itself only enables intra-PE parallelism and requires an additional module for inter-PE communication.

### 2.2 HClib-Actor

HClib-Actor [11] is an external module for HClib that enables the FA-BSP execution. It offers an SPMD-style programming, maintaining interoperability with

existing MPI and OpenSHMEM applications. Specifically, in any superstep that could benefit from FA-BSP execution, the user can leverage the HCLib-Actor API to enable fine-grained point-to-point asynchronous messaging with actor/s-selector<sup>2</sup>. Because this execution model inherently produces many fine-grained messages, the runtime automatically performs message aggregation for better network utilization, which is backed by the Conveyors library [10].

Listing 1.1 and Listing 1.2 demonstrate an FA-BSP program. In this program, each processing element (PE) sends  $N$  messages to arbitrary destinations, incrementing a target element of a remote array by one. In Listing 1.1, each PE first allocates a local array `larray` (Line 2). Second, each PE instantiates an actor instance (Line 3). Third, each PE starts the actor (Line 6) and sends  $N$  asynchronous messages to random destinations (Line 10). The `done` API (Line 12) is used to inform the runtime that the current PE will not send any more messages so as to aid the runtime with overall application termination. The code in Listing 1.2 defines an actor class that includes the message handler (Line 5). It is important to note that no atomics are required on Line 6 when updating `larray` because the runtime processes incoming messages one at a time.

Each PE is single-threaded, and the runtime executes the portions in an interleaved fashion. It is important to note that the `finish` construct acts as a bulk synchronization construct, even with asynchronous messaging. This means it waits for all outgoing messages to be sent, all incoming messages to be processed, and all tasks spawned within the `finish` scope to be completed.

**Listing 1.1.** The Main Part (computation and asynchronous messaging in Figure 1). **Listing 1.2.** The Handler Part (Message handling in Figure 1).

```

1 // SPMD
2 int* larray = (int*)calloc(N, sizeof(int));
3 MyActor* actor_ptr = new MyActor(larray);
4 // one superstep
5 hclib::finish([=]() {
6     actor_ptr->start();
7     for (int i = 0; i < N; i++) {
8         int dst = ...;
9         // Asynchronous SEND
10        actor_ptr->send(i, dst);
11    }
12    actor_ptr->done(0);
13 });
14 // barrier synchronization/collective

```

```

1 // Actor Class
2 class MyActor: public hclib::Selector
3 {
4     <1, int> {
5     int *larray;
6     // Message Handler
7     void process(int idx, int
8         sender_rank) {
9         larray[idx] += 1; // no
10        atomics
11    }
12 }
13 public:
14 MyActor(int *larray) : larray(
15     larray) {
16     mb[0].process = [this](int idx
17         , int sender_rank) {
18         this->process(idx,
19             sender_rank);
20     };
21 }
22 };
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

### 3 Preliminary Design

The primary goal of this paper is to enable asynchronous tasking in message handlers. Since HCLib-Actor already inherits tasking API from HCLib, we look to

<sup>2</sup> Selector [9] is an actor with multiple mailboxes.

enable asynchronous tasking in a way that is natural to both HCLib and HCLib-Actor programmers.

As with the original HCLib, `async` is used to create an asynchronous task within a message handler, and also `finish` is used to block until all children tasks created within it are completed.

Currently, we support the following types of messages:

- **Blocking Messages** (Listing 1.3): The original actor model mandates that each actor processes incoming messages one at a time. To comply with it, the user must use `finish` appropriately so any tasks initiated within a message handler must be completed before it is completed.
- **Non-blocking Messages** (Listing 1.4): This type of message is intended to relax the message processing rule in order to achieve performance enhancements. Such a message can create an escaping task that cannot be guaranteed to be completed until `finish` in the main process has been unblocked (Line 5 in Listing 1.1). Since multiple tasks can be executed simultaneously, the user needs to eliminate data race across all invocations of any message handlers, and the main part.

**Listing 1.3.** Blocking message with **Listing 1.4.** Non-blocking message.

```
finish.
1 // Message Handler
2 void process(int idx, int sender_rank) {
3   hclib:\, \!:finish( [= ] {
4     hclib:\, \!:async( [= ] { ... } ); // T1
5   });
6 }
```

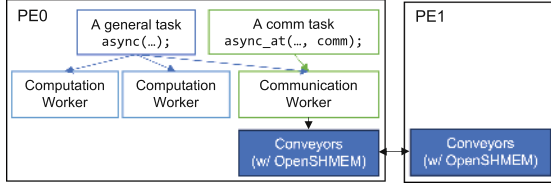
```
1 // Message Handler
2 void process(int idx, int sender_rank) {
3   hclib::async( [= ] { ... } ); // T2
4   // no synchronization (non-blocking)
5   // until "finish" in the main part
6 }
```

To summarize, using blocking messages prevents data races as long as `finish` and `async` are used properly and data races within a message handler are eliminated, whereas using non-blocking messages requires careful data race elimination because multiple messages can be processed simultaneously.

## 4 Prototype Runtime Implementation

A key challenge in enabling user-level asynchronous tasking in the runtime is that the Conveyors API routines are not thread-safe. As opposed to the original single-threaded execution model (see Sect. 2.2), the runtime must ensure that only one worker thread can communicate with Conveyors. `async_at` is suitable for achieving this because it can constrain which worker can execute a specific task [7, 14]. As shown in Fig. 2, a communication task is created via `async_at` and can only be executed by the communication worker, while a general task can be executed by any worker.

While the use of `async_at` is sufficient in most parts of the runtime, there is a non-trivial problem in the implementation of `finish`. In the original HCLib implementation, the runtime creates an asynchronous task with the continuation of `finish` and tries to schedule another pending task so that a specific worker will not be blocked. Since the original implementation allowed a non-communication worker to execute the continuation, which can invoke Conveyors routines, we modified the `finish` implementation so that the continuation task is always executed by the communication worker.



**Fig. 2.** The execution model of the FA-BSP runtime with asynchronous tasking enabled.

## 5 Case Study with Distributed Triangle Counting

This section discusses triangle counting implementations with user-level asynchronous tasking and results of an empirical evaluation on a multi-node platform.

### 5.1 Distributed Triangle Counting with the FA-BSP Model

Triangle Counting counts all possible numbers of triangles in a graph. Our baseline implementation is the one in the FA-BSP paper [11], where each processing element (PE or actor) iterates over the neighbors of each local vertex ( $v_i$ ) that resides on the actor and finds two different neighbors (vertices  $v_j$  and  $v_k$ ) and sends a message to a (possibly) remote actor that owns  $v_j$ . The receiver receives a pair  $(v_j, v_k)$  and checks if there is an edge  $v_j \rightarrow v_k$ . The edge check can be performed using binary search on a neighbor list of  $v_j$ .

While a binary search can be parallelized, we aim to increase the granularity of the message handler to amortize the cost of task creation. Specifically, we group multiple pairs into a single big packet and process them in parallel in the message handler. We call it *a chunked version*.

### 5.2 Experimental Setup

We perform our experiments on the CPU nodes of the PACE cluster at Georgia Tech<sup>3</sup>. Each CPU node has Dual Intel Xeon Gold 6226 CPUs with 24 total physical cores and 192 GB of DDR4 memory connected by an Infiniband 100HDR interconnect. For the software stack, we use `gcc/10.3.0` and `openmpi/4.1.4`.

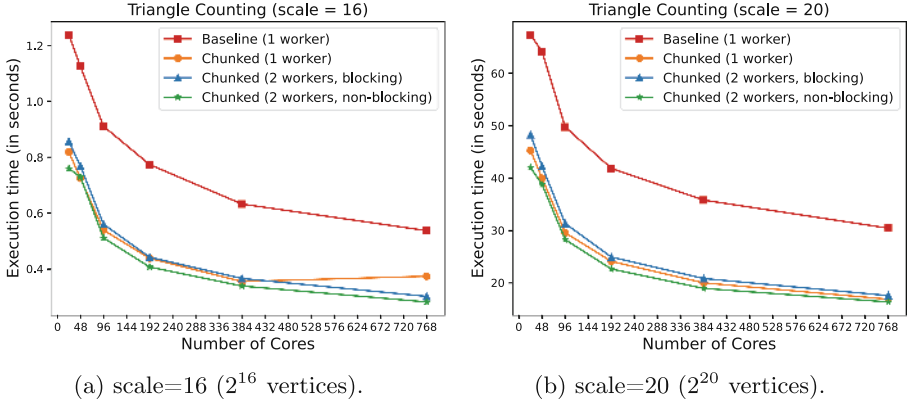
We compare the performance of the following three variants while keeping the number of cores constant ( $c$ ):

1. **Baseline:** The original version discussed in Sect. 5.1. We ran it with  $c$  PEs, where  $c$  is the total number of cores.
2. **The chunked version with a single worker:** The chunked version discussed in Sect. 5.1 with a chunk size of 624 pairs (16 bytes/pair). As with the baseline version, we ran it with  $c$  PEs, where  $c$  is the total number of cores. There is no asynchronous tasking in the user application.

<sup>3</sup> <https://www.pace.gatech.edu/>.

**3. The chunked with multiple workers:** The chunked version discussed in Sect. 5.1 with the same chunk size. We ran the application with  $c/w$  PEs, where  $w$  is the number of worker threads per PE. This variant also has two sub-variants: blocking and non-blocking message versions (Sect. 3). To minimize task creation overhead, the 624 pairs are divided into  $w$  chunks.

These variants were run with scale 16 and 20 graphs, the R-MAT parameters of  $A = 57.0$ ,  $B = C = 19.0$ , and  $D = 5.0$ , and an edge factor of 16. We report the best of five measurements.



**Fig. 3.** Strong scaling results of Triangle Counting on the PACE cluster with up to 32 nodes and 768 cores (absolute timings, lower is better).

### 5.3 Preliminary Results

Figure 3 (a)(b) show strong scaling results of Triangle Counting. The results show that the chunked version significantly outperforms the baseline version even in the single-worker setting because this enables better network utilization even with the *Conveyors* aggregation library. Moreover, the single-worker variant is 1) faster than the blocking version of the two-worker variant but 2) slower than the non-blocking version of the two-worker variant in both data sizes. In summary, the non-blocking version achieves 3–32% performance improvements compared to the chunked single-worker version.

Although, we do observe that using more than 2 workers results in a noticeable drop in performance due to the increase in task creation overhead as the number of workers increases since it creates  $w - 1$  tasks. In general, tweaking the chunk size and the number of workers can further improve/degrade the performance. However, our goal in this paper is to find a case where asynchronous tasking is beneficial, and tuning these parameters for the best performance is beyond the scope of the paper and left as future work.



## 6 Conclusion and Future Work

This paper explores the exploitation of asynchronous tasking in the FA-BSP model. We enhance HCLib-Actor to support task parallel constructs like `async` and `finish` within a message handler and discussed a case where user-level asynchronous tasking is beneficial. Results show that asynchronous tasking leads to 3% to 32% performance improvements. In future work, we plan to explore more opportunities for parallelization in different graph applications as well as the introduction of parallelizing constructs within the FA-BSP runtime.

**Acknowledgments.** This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0083. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government. The authors would also like to thank Professor Vivek Sarkar of the College of Computing, Georgia Institute of Technology, for his support of this work.



## References

1. Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA (1986)
2. Chapman, B., et al.: Introducing OpenSHMEM: SHMEM for the PGAS community. In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model. PGAS '10*, Association for Computing Machinery, New York, NY, USA (2010). <https://doi.org/10.1145/2020373.2020375>, <https://doi.org/10.1145/2020373.2020375>
3. Charles, P., et al.: X10: an object-oriented approach to non-uniform cluster computing. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 519–538. OOPSLA '05, Association for Computing Machinery, New York, NY, USA (2005)<https://doi.org/10.1145/1094811.1094852>
4. Elmougy, Y., Hayashi, A., Sarkar, V.: Highly scalable large-scale asynchronous graph processing using actors. In: *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*, pp. 242–248 (2023). <https://doi.org/10.1109/CCGridW59191.2023.00049>
5. Elmougy, Y., Hayashi, A., Sarkar, V.: A distributed, asynchronous algorithm for large-scale internet network topology analysis. In: *2024 IEEE/ACM 24th International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)* (2024)
6. Elmougy, Y., Hayashi, A., Sarkar, V.: Asynchronous distributed actor-based approach to jaccard similarity for genome comparisons. In: *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, pp. 1–11 (2024). <https://doi.org/10.23919/ISC.2024.10528922>

7. Grossman, M., Kumar, V., Vrvilo, N., Budimlic, Z., Sarkar, V.: A pluggable framework for composable hpc scheduling libraries. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017, Orlando / Buena Vista, FL, USA, May 29 - June 2, 2017, pp. 723–732. IEEE Computer Society (2017). <https://doi.org/10.1109/IPDPSW.2017.13>
8. Imam, S.M., Sarkar, V.: Integrating task parallelism with actors. SIGPLAN Not. **47**(10), 753–772 (Oct 2012). <https://doi.org/10.1145/2398857.2384671>
9. Imam, S.M., Sarkar, V.: Selectors: actors with multiple guarded mailboxes. In: Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control, pp. 1–14. AGERE! '14, Association for Computing Machinery, New York, NY, USA (2014). <https://doi.org/10.1145/2687357.2687360>
10. Maley, F.M., DeVinney, J.G.: Conveyors for streaming many-to-many communication. In: 2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3), pp. 1–8 (2019) <https://doi.org/10.1109/IA349570.2019.00007>
11. Paul, S.R., Hayashi, A., Chen, K., Elmougy, Y., Sarkar, V.: A fine-grained asynchronous bulk synchronous parallelism model for PGAS applications. J. Comput. Sci. **69**, 102014 (2023). <https://doi.org/10.1016/j.jocs.2023.102014>
12. Singhal, S.P., Hati, S., Young, J., Sarkar, V., Hayashi, A., Vuduc, R.: Asynchronous distributed-memory parallel algorithms for influence maximization. In: 37th International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (2024)
13. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (Aug 1990). <https://doi.org/10.1145/79173.79181>
14. Yan, Y., Zhao, J., Guo, Y., Sarkar, V.: Hierarchical place trees: a portable abstraction for task parallelism and data movement. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 172–187. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-13374-9\\_12](https://doi.org/10.1007/978-3-642-13374-9_12)



# Learning to Harness In-Vitro Biological Neural Networks

Frithjof Gressmann<sup>(✉)</sup>  and Lawrence Rauchwerger 

Department of Computer Science, University of Illinois at Urbana-Champaign,  
Urbana, IL 61801, USA  
`{fg14,rwerger}@illinois.edu`

**Abstract.** Advancements in bio-engineering have enabled the creation of in-vitro biological neural networks, offering an exciting avenue for a new kind of computational platform. A computing stack powered by living neurons could unlock self-organizing and dynamically rewiring systems with extreme connectivity and parallel processing power, all while running on sugar with unprecedented energy efficiency. Despite their potential, computing applications of these biological systems remain a nascent and limited technology that presents a challenging and radical departure from the precise, digital von Neumann architectures that dominate today’s computing landscape. Here, we outline a framework that leverages *in-silico* simulation to establish an engineering testbed with the ultimate goal of learning to harness neural *in-vitro* systems for computational purposes. We describe an optimization approach to uncover reproducible neural activity present in a system that can be leveraged to carry out basic information processing tasks. We demonstrate the feasibility of this approach by optimizing a simulated neural system to perform digit classification, offering a proof-of-concept for a potential pathway to leveraging neural computation in vitro.

**Keywords:** Machine Learning · Neuromorphic computing · Biological neural networks

## 1 Introduction

The stunning success of deep neural networks in machine learning combined with the slowing of Moore’s law is spurring interest in novel, brain-inspired computing approaches that could bring about next-generation high-performance, low-power architectures [16–18]. However, today’s digital, CMOS architectures are still limited in their ability to process complex, unstructured, and noisy data with the extreme energy efficiency of their biological counterparts. Recently, advances in bio-engineering have opened up new possibilities through the construction of biological neural networks *in-vitro*, offering not only a novel “wetware” substrate for computing but also a vehicle for gaining a deeper understanding of neural processing systems akin to the brain [3, 5, 21, 25]. These engineered living biological networks may ultimately emerge as an alternative hardware for artificial

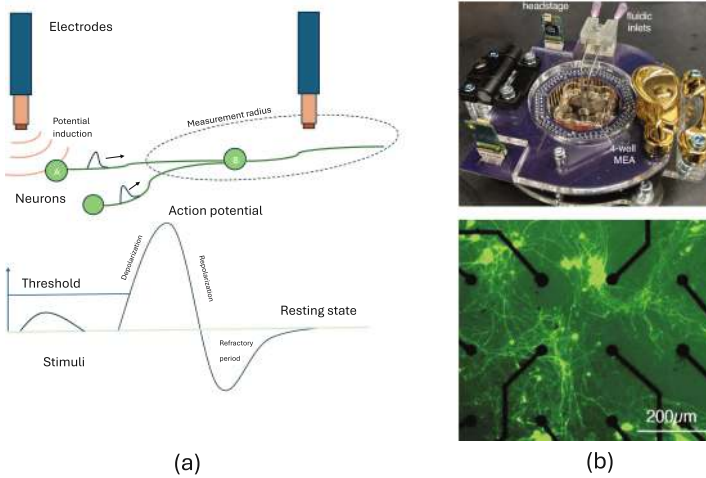
intelligence applications, a development that would make the quest for intelligent compute come full circle. Despite these promising prospects, engineering cell cultures for computing applications remains in its infancy. Current platforms offer only crude input control, limited temporal and spatial resolution of measured resulting activity, and under-characterized and insufficiently understood computing properties. In particular, how to effectively design, program, and leverage the systems for computation is an open question and active area of research [21].

Notably, interacting with living neural networks for computing purposes poses two fundamental challenges. First, while the basic physiological mechanisms that drive neurons have been uncovered, we do not know how these processes give rise to the remarkable computing capabilities of neural systems. To make an analogy to conventional processors, having understood the basic physics behind a transistor, we still do not know how a complex composition of transistors could implement higher-level logic and algorithms [11]. Secondly, current experimental methods do not offer enough precision to measure and manipulate all potentially relevant neuronal processes, especially in larger cell cultures. In practice, input-output interfaces remain limited to crude interventions that are hard to calibrate and target precisely. Despite these challenges, a growing body of work suggests that living neuronal systems may be leveraged without a complete understanding or command of their neural dynamics. The field has seen remarkable practical achievements in interacting with neural systems, enabling, for instance, decoding of thought through brain-computer interfaces [8], induced motor control in simple organisms [13], or video game play using neural feedback [12]. A key ingredient to these successes has been data-driven learning and analysis methods that can build *implicit* representations of the neural dynamics and enable systematic optimization towards desirable states and dynamics. For example, algorithmic data analysis of neural data can help fill knowledge gaps and automatically uncover functional and structural properties of neural systems [20].

In this work, we present a simulation-driven approach that is designed to discover neural stimulation patterns that induce reproducible neural activity in vitro. We develop an in-silico simulation that recapitulates key features of the open in-vitro experimentation platform by [24] (see Fig. 1). Using a contrastive training approach, we demonstrate as a proof of concept that it is possible to exploit the neuronal dynamics of such a system for a basic classification task. This presents a first step towards a general approach to induce and control neural activity for downstream tasks, that may ultimately enable a harnessing of neural computation in the corresponding real-world in-vitro system.

## 2 Background

Unlike their artificial counterparts, biological neurons exchange information using action potentials – short-lived changes in the membrane potential also known as *spikes* that travel along axonal connections to communicate with the



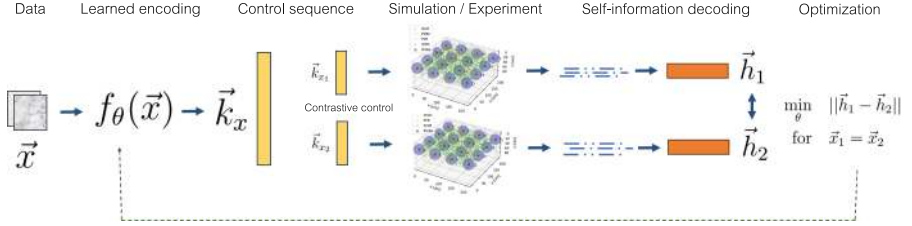
**Fig. 1.** (a) Illustration of spiking dynamics and interaction with recording and stimulation electrodes. Stimuli raise the action potential of a neuron causing it to depolarize if the firing threshold is crossed. The electrodes pick up changes in the extra-cellular potential, allowing the detection of spiking activity of nearby neurons. (b) Recording system by Zhang et al. [24] with fluorescence microscopic image of seeded cells on MEA [Photos by [24]/CC BY-NC-ND 4.0]

connected cells (see Fig. 1a). When neuron A transmits a spike to neuron B, its cell potential increases and moves closer to a spiking threshold, before decaying back away to the resting potential. If multiple spikes arrive in a sufficiently short time window, however, the threshold is crossed causing the neuron to produce another spike. Experimentally, it is possible to interact with this process by measuring and manipulating the cellular potentials, for example, by inducing currents that raise cell potentials beyond the firing threshold.

Historically, the progress in understanding neurophysiology has been driven by experiments *in vivo* [7]. More recently, however, advancements in biological technology have opened up the creation and study of neural systems *in vitro*, outside of their natural biological context. Notably, the groundbreaking development of induced pluripotent stem cell technology (iPSC) has enabled the reprogramming of human cells to stem cells and revolutionized the field [22]. Importantly, it has brought about the techniques that are required to grow brain-like cell cultures of increasing sophistication in lab environments ([25] provides an overview of this “organoid” technology). At the same time, neural recording and stimulation technology has been steadily improving and is enabling increasingly high-resolution electrophysiological measurements with minimal disruption to the cell tissues [2]. One of the most common stimulation and recording devices is multi-electrode arrays (MEAs) that consist of multiple microelectrodes from which neural signals can be picked up or delivered. While electrodes have a limited measurement radius and pick up the combined signal of all electrical activity in the

neighborhood, post-processing methods can “sort” the activity and uncover the underlying neuronal sources with a high degree of accuracy [19]. Taken together, this creates an experimental platform to explore neural responses to a wide range of input stimuli and ultimately their computational capabilities [21].

### 3 Approach



**Fig. 2.** Framework overview. To learn control function  $f$  parameterized by  $\theta$  that outputs a control sequence  $\mathbf{k}$  for a given data sample  $\mathbf{x}$ , an input is processed multiple times and corresponding neural activity for the trials is recorded. For a given input pair, embedding vectors  $\mathbf{h}_1$  and  $\mathbf{h}_2$  are computed by analyzing the self-information content of the activity. Finally,  $\theta$  is optimized in a contrastive fashion by minimization of the embedding vector distance of positive samples  $\mathbf{x}_1 = \mathbf{x}_2$  while maximizing the distance of negative samples  $\mathbf{x}_1 \neq \mathbf{x}_2$ .

At a high level, the problem of leveraging a given in-vitro system can be broken down into three sub-problems.

- i How to encode data such that it can be fed into the experimental platform,
- ii how to decode resulting neural activity, and
- iii how to establish some level of control over the resulting input-output system.

In this work, we propose an approach that addresses (i)-(iii) in a way that allows us to optimize the system end-to-end (see Fig. 2).

#### 3.1 (i) Learned Control Sequence Model

Experimental systems that interface with in-vitro cell cultures can vary dramatically in terms of used technology and capability (see Sect. 2). However, typically, experimental interactions come down to specifying *stimulation* of the available input channels at certain *times*. For example, this may be a list of times when to activate a laser light that shines on the culture, or a list of times when to induce a current at an electrode placed somewhere in the culture. In this setting, system control can be described as a mapping  $f : \mathbb{R}^{N \times T} \rightarrow \mathbb{R}^{j(x) \times T'}$  that transforms given  $N$ -dimensional data into a sequence of stimulation times. The goal is to find some parameters  $\theta$  such that  $f_\theta(x) = \mathbf{k}$  controls the neural system in some desirable way.

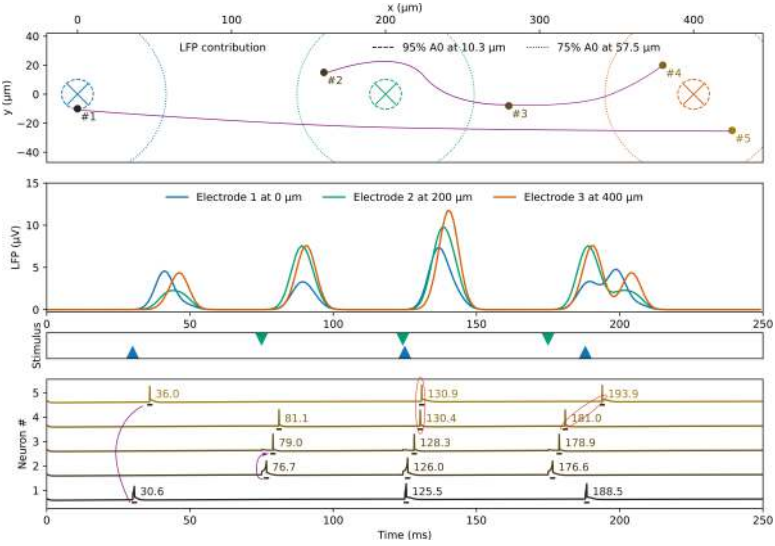
### 3.2 (ii) Self-information Decoding

While it is clear that neurons leverage the timing of action potentials as main carrier of information, it is not known how exactly information is encoded in the spike times [6]. Many candidate coding schemes exist [1] and there is increasing empirical evidence that individual spike times rather than averaged firing rates over extended time windows play a crucial role in the neural code [23]. However, without averaging, the enormous variability with which neurons elicit spikes poses a practical challenge [14]. How can information be reliably decoded from inherently noisy and stochastic neural processes? This question is particularly relevant in the context of MEA-based in-vitro architectures where the electrode measurements are subject to additional noise and imperfections. Addressing this problem, Li and Tsien have proposed a self-information decoding scheme in which the neural variability itself is used for coding rather than regarded as mere noise [14]. Specifically, the idea is to look at the probability distribution of the time between observed spikes (inter-spike intervals, ISI) and ask how likely the currently observed silence duration is given the past ‘ground state’ of variability. In other words, how surprising is the observed silence duration between two spikes, either by being extremely short or unexpectedly prolonged compared to the past distribution. The overly short or long states then carry information in contrast to the likely inter-spike intervals that are close to the mean of the variability distribution. In an experimental in-vivo study involving mice, Li et al. have demonstrated that self-information decoding could uncover cell assemblies active in response to induced cognitive states like fear, suggesting that the self-information principle can be a practically effective decoding strategy [15]. While the neural self-information theory warrants further experimental scrutiny, one immediate and general utility of the idea is that the method can uncover a broad scope of unexpected states in an unbiased manner. In practice, we can leverage this to identify meaningful neural activity without having to predefine a reference point set by an outside observer. As a result, the decoding scheme may provide enough flexibility for the control sequence model to uncover above-the-noise states without pre-imposing the exact character of such states.

Practically, to compute self-information from the measured neural activity of the in-vitro system, we record the intervals between the points in time when spiking activity is detected. We then estimate the probability density function (PDF) of the intervals using Gaussian kernel density estimation, denoted as  $p(x)$ . The self-information is defined as  $I_X(x) := -\log[p_X(x)]$  where  $X$  represents the random variable of intervals. This approach provides a non-parametric estimate of the information content in the spiking activity based on its temporal structure. We can obtain this estimate for all recording positions  $j$  in the cell culture, yielding a vector  $\mathbf{h}$  with elements  $I_{X(j)}(x)$ .

Taken together, it is important to note that this self-information embedding  $\mathbf{h}$  captures valuable information about the *spatial-temporal* activity in the culture in response to stimulation patterns. Figure 3 illustrates this in a toy system consisting of three spatially distributed recording channels and five neurons that respond to varying stimulation patterns. Consider that channel C picks up the

activity of neurons #4 and #5 in its neighborhood. Since #4 is connected to neurons in the neighborhood of channel B and #5 is connected to a neuron in the neighborhood of channel A, stimulating A and B in isolation brings about average activity in C. However, when A and B are stimulated in quick succession, C will observe an unexpectedly short inter-spike interval. Moreover, the delay between stimulation of A and B matters and has to account for the longer signal traveling time from A to C versus B to C. While this is a highly idealized example, it illustrates how stimulation response activity retains information about the spatio-temporal structure of culture. Note that this is true even though neither the neuron positioning nor their connectivity are directly observable by the electrodes. Given enough sample data, it would be possible to infer how to stimulate the system to induce events with a high information content.



**Fig. 3.** Toy example system with three electrodes A, B, and C that can stimulate and record the resulting spiking activity and local field potential (LFP) of neurons in their neighborhood. Under the right stimulation pattern of A and B, electrode C observes an unexpectedly short inter-spiking interval and high LFP contribution at 130 ms, an event containing high spatiotemporal information content.

### 3.3 (iii) Contrastive Optimization

The self-information decoding translates multi-channel activity in a given time window into an embedding vector  $\mathbf{h}$  that quantifies the spatiotemporal unexpectedness of the observed activity. This gives us a way to compare and order states using an embedding vector norm  $\|\mathbf{h}_1 - \mathbf{h}_2\|$ . In this setup, a minimal norm implies a similar level of information content or unexpectedness of the two



compared states. We can exploit this property to learn to distinguish stimulus-response patterns by recording and comparing neural activity shortly after stimulation. Figure 2 illustrates this process. Given two data samples  $\mathbf{x}_1$  and  $\mathbf{x}_2$  we can compute the corresponding control sequences  $\mathbf{k}_{1|2} = f(\mathbf{x}_{1|2})$  that describe the stimulation pattern to apply to the system. The resulting embedding norm should be minimal if  $\mathbf{x}_1 = \mathbf{x}_2$  and maximal otherwise. In other words, the same stimulus applied multiple times should give responses of similar self-information content. This is reminiscent of the contrastive loss function used for deep metric learning [4]. Note that optimizing  $f_\theta$  to conform to this condition does not specify whether the response self-information of  $\mathbf{x}_{1|2}$  should be high or low, just that they should be different. Another way to think of this optimization strategy is as a search for a set of points in the stimulation space that have predictable, low-information responses, as opposed to another set of points that lead to surprising, high-information responses of the neurons.

### 3.4 Learning to Harness Neural Computation

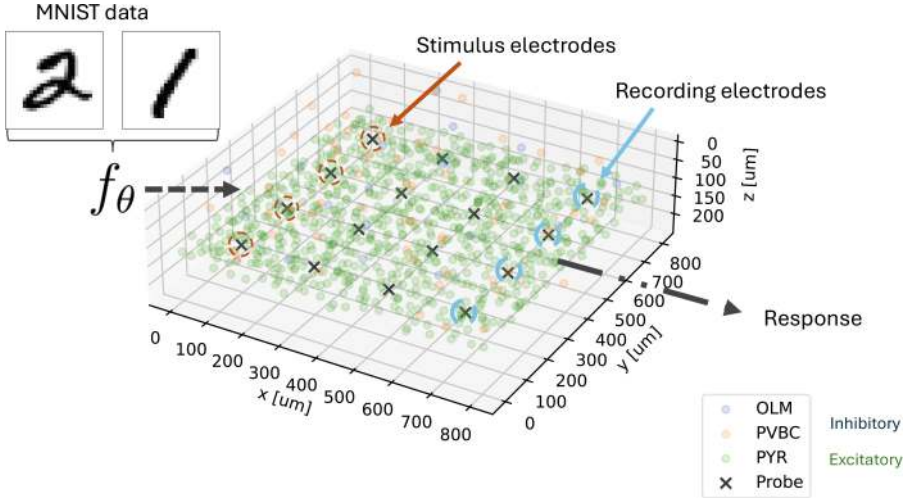
Crucially, uncovering the relationship between input stimulation and neural response provides a pathway to leveraging the neural system for computing tasks. To illustrate this, suppose that we have figured out that two stimulation patterns  $\alpha$  and  $\beta$  lead to clearly distinguishable neural responses  $A$  and  $B$ . We could leverage this separation by mapping pictures of cats to patterns like  $\alpha$  and pictures of dogs to patterns like  $\beta$ . We could then map an unknown image in the same way and interpret the response  $A$  as a recognition of a cat and  $B$  as a recognition of a dog. This “computation” can be more powerful than using the simple mapping function directly since the neural system already implements a powerful separation function that is robust to noise and processes the relevant features of the input signal. We can directly incorporate this mapping strategy into the training process by adjusting the equality condition  $\mathbf{x}_1 = \mathbf{x}_2$  with  $y_1 = y_2$  where  $y_{1|2}$  are the class labels for  $\mathbf{x}_{1|2}$  (e.g. “dog” and “cat” for a dog and cat image). This adjustment means that  $f_\theta$  encodes images into stimulation patterns such that the neural response is different for images with a different class. More generally, the framework thus allows us to find suitable stimulation patterns that encode a given dataset such that distinct data samples elicit distinguished neural responses.

## 4 Experiments

Testing an approach like the one outlined above in an in-vitro system poses many practical challenges. First and foremost, designing and conducting experiments in vitro is costly and time-intensive. Neural cultures take weeks to grow and effort to keep alive. Moreover, no culture is ever the same and evolves with age, making it harder to control for spurious changes and verify results in repeated trials. To accelerate the experimental design, it is thus a good idea to experiment with algorithmic approaches in simulation first. This allows for faster iteration

but also provides a way to inspect and “debug” neural dynamics without measurement limitations. Here, we implement a neural simulation that recapitulates key properties of a contemporary in-vitro experimental platform as a testbed for algorithmic approaches to leveraging neural computation. We demonstrate *in-silico* that it is possible to find an  $f_\theta$  to perform digit classification via multi-electrode array stimulation and recording.

#### 4.1 Neural Simulation



**Fig. 4.** Visualization of the simulated in-vitro system. The neuron positions of the different cell types in color are not drawn to size. The black crosses mark the position of the electrodes to stimulate and record from nearby neurons. (Color figure online)

The simulated system is depicted in Fig. 4 and aims to recapitulate the real-world in-vitro experimentation platform by Zhang et al. [24] (see Fig. 1b). It consists of a  $4 \times 4$  multi-electrode array with a pitch of  $200\mu\text{m}$  placed on top of a neural culture with excitatory and inhibitory neurons. We adopt the electrode model provided by Johnsen et al. [10] and assume that each electrode can perfectly detect spikes in a  $50\mu\text{m}$  radius, dropping to 50% detection within a  $100\mu\text{m}$  radius. For stimulation, we employ a simplified approach to estimate the induced voltage  $V_{induced}$  in neurons that are within  $150\mu\text{m}$  of the electrode:

$$V_{induced} = \frac{V_{stim}}{r} \cdot d_{neuron} \cdot \cos(\gamma)$$

where  $V_{stim}$  is the stimulation voltage of 750 mV,  $r$  is the distance from the electrode,  $d_{neuron}$  is the neuron diameter assumed to be  $10\mu\text{m}$ , and  $\gamma$  is the angle

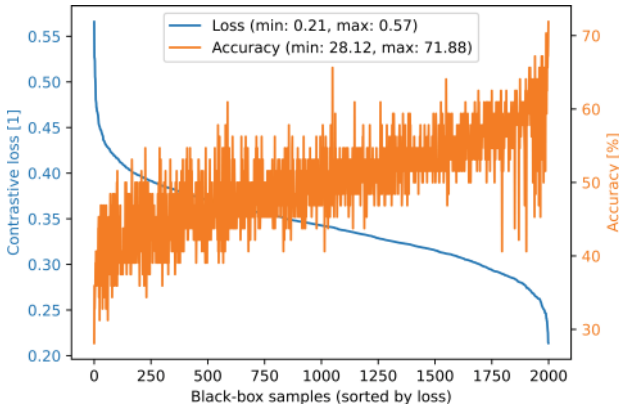
between the electric field and the neuron dipole orientation assumed to be  $0^\circ$ . The neurons are distributed randomly and connected with a probability that is proportional to their spatial distance. The neuron dynamics are modeled using an Izhikevich 2003 model [9] that implements biologically plausible dynamics with high computational efficiency. Synaptic connections are tuned such that the culture dynamics produce the empirical firing rates reported in [24].

## 4.2 Optimization

The system is stimulated using the four bottom electrodes in the MEA within a 50 ms window (Fig. 4). The time scale of this recording window is somewhat arbitrary but chosen to be in the order of magnitude of time that multiple actions potentials take to travel through the culture. Activity is recorded from the four top electrodes only. This ensures that the input stimulation area does not overlap with the output area, i.e. measured activity has been transmitted through the neural dynamics. The control sequence model  $f_\theta$  is implemented by a feed-forward artificial neural network that outputs the stimulation times for the four input channels. We use the well-known MNIST digit dataset as a benchmark and downsample the  $28 \times 28$  image to  $3 \times 3$  pixels to reduce the number of learnable parameters  $|\theta|$  to 450. We minimize using black-box optimization with the loss function

$$\mathcal{L}(\mathbf{x}_i, \mathbf{x}_j, \theta) = \mathbb{1}[y_i = y_j] \|\mathbf{k}_\theta(\mathbf{x}_i) - \mathbf{k}_\theta(\mathbf{x}_j)\|_2^2 + \mathbb{1}[y_i \neq y_j] \max(0, 1 - \|\mathbf{k}_\theta(\mathbf{x}_i) - \mathbf{k}_\theta(\mathbf{x}_j)\|_2)^2$$

where  $\mathbf{x}_{i|j}$  are the data samples with class labels  $y_{i|j}$  and  $\mathbf{k}$  are the self-information embedding vectors of the neural activity (see Sect. 3).



**Fig. 5.** Contrastive MNIST digit optimization: loss and accuracy for 2000 random  $\theta$ -samples sorted by loss. Accuracy is determined on 64-digit pairs from the training set, where guessing would yield a 50% accuracy. Best  $\theta$  achieves an accuracy of 72%, suggesting that it is possible to find control parameters that classify the digits with an accuracy above chance.

Figure 5 shows the optimization objective  $\mathcal{L}$  computed for 2000 simulations with  $\theta$ -values that are sampled via Latin hypercube sampling. Each  $\theta$  is evaluated on 64 MNIST digit pairs by computing the control sequence and evaluating the prediction accuracy of whether the given digits belong to the same class. The best-performing parameter configuration achieves an accuracy of over 70%, outperforming the random guessing baseline of 50%. This result suggests that it is possible to find a  $\theta$  that encodes the images of digits into stimulation patterns that induce distinct neural responses for different digits with an above-chance accuracy. Importantly, this works despite the noisy dynamics that are never quite the same at the time of the electrode stimulation.

## 5 Conclusion

We have presented a simulation-driven approach to learn stimulation patterns that steer in-vitro neural activity towards desired responses. We demonstrated in simulation that the strategy can be used to exploit the system dynamics to perform a basic classification task. While this presents a first step, crucial work remains to demonstrate the effectiveness of the approach in a real-world system. For one, we have resorted to a naive black-box optimization approach of trying random parameter changes and selecting for lower loss values. This is not efficient nor scalable and could be improved in multiple ways. Gradient-free optimization methods such as various evolutionary optimization strategies could be employed for more effective optimization. It may also be possible to leverage domain knowledge to construct differentiable approximations of the system to leverage more effective gradient-based search methods. Secondly, it will be important to validate simulation results against real-world data to test the eventual applicability in lab experiments. Notably, lab data can provide important information on what properties are the most important to get right when simulating plausible neural dynamics. Finally, in the scope of this work, we have deliberately ignored the remarkable malleability of neural systems that could be used to “program” rather than merely exploit already existing behaviors. This could become another optimization target in future work. It is plausible that a further improvement of simulation and optimization capabilities, guided by feedback from real-world experiments, will pave the way to increasingly sophisticated computing applications in vitro.

## References

1. Auge, D., Hille, J., Mueller, E., Knoll, A.: A survey of encoding techniques for signal processing in spiking neural networks. *Neural Process. Lett.* **53**(6), 4693–4710 (2021). <https://doi.org/10.1007/s11063-021-10562-2>
2. Chen, R., Canales, A., Anikeeva, P.: Neural recording and modulation technologies. *Nat. Rev. Mater.* **2**(2), 1–16 (2017). <https://doi.org/10.1038/natrevmats.2016.93>
3. Chen, Z., Liang, Q., Wei, Z., Chen, X., Shi, Q., Yu, Z., Sun, T.: An overview of in vitro biological neural networks for robot intelligence. *Cyborg Bionic Syst.* **4**, 0001 (2023). <https://doi.org/10.34133/cbsystems.0001>

4. Chopra, S., Hadsell, R., LeCun, Y.: Learning a similarity metric discriminatively, with application to face verification. In: 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), vol. 1, pp. 539–546 (2005). <https://doi.org/10.1109/CVPR.2005.202>
5. Chow, S., Hu, H., Osaki, T., Levi, T., Ikeuchi, Y.: Advances in construction and modeling of functional neural circuits in vitro. *Neurochem. Res.* **47**(9), 2529–2544 (2022). <https://doi.org/10.1007/s11064-022-03682-1>
6. Eggermont, J.J.: Is there a neural code? *Neurosci. Biobehav. Rev.* **22**(2), 355–370 (1998). [https://doi.org/10.1016/S0149-7634\(97\)00021-3](https://doi.org/10.1016/S0149-7634(97)00021-3)
7. Finger, S.: *Origins of Neuroscience: A History of Explorations into Brain Function*. Oxford University Press, New York (1994)
8. Gao, X., Wang, Y., Chen, X., Gao, S.: Interface, interaction, and intelligence in generalized brain-computer interfaces. *Trends Cogn. Sci.* **25**(8), 671–684 (2021). <https://doi.org/10.1016/j.tics.2021.04.003>
9. Izhikevich, E.: Simple model of spiking neurons. *IEEE Trans. Neural Networks* **14**(6), 1569–1572 (2003). <https://doi.org/10.1109/TNN.2003.820440>
10. Johnsen, K.A., Cruzado, N.A., Willats, A.A., Rozell, C.J.: Cleo: A testbed for bridging model and experiment by simulating closed-loop stimulation, electrode recording, and optogenetics (2023). <https://doi.org/10.1101/2023.01.27.525963>
11. Jonas, E., Kording, K.P.: Could a neuroscientist understand a microprocessor? *PLoS Comput. Biol.* **13**(1), e1005268 (2017). <https://doi.org/10.1371/journal.pcbi.1005268>
12. Kagan, B.J., et al.: In vitro neurons learn and exhibit sentience when embodied in a simulated game-world. *Neuron* **110**(23), 3952–3969.e8 (2022). <https://doi.org/10.1016/j.neuron.2022.09.001>
13. Li, C., Kreiman, G., Ramanathan, S.: Discovering neural policies to drive behaviour by integrating deep reinforcement learning agents with biological neural networks. *Nat. Mach. Intell.* **6**(6), 726–738 (2024). <https://doi.org/10.1038/s42256-024-00854-2>
14. Li, M., Tsien, J.Z.: Neural code—neural self-information theory on how cell-assembly code rises from spike time and neuronal variability. *Front. Cell. Neurosci.* **11** (2017). <https://doi.org/10.3389/fncel.2017.00236>
15. Li, M., et al.: Neural coding of cell assemblies via spike-timing self-information. *Cereb. Cortex* **28**(7), 2563–2576 (2018). <https://doi.org/10.1093/cercor/bhy081>
16. Mead, C.: Neuromorphic electronic systems. *Proc. IEEE* **78**(10), 1629–1636 (1990). <https://doi.org/10.1109/5.58356>
17. Mead, C.: How we created neuromorphic engineering. *Nat. Electron.* **3**(7), 434–435 (2020). <https://doi.org/10.1038/s41928-020-0448-2>
18. Monroe, D.: Neuromorphic computing gets ready for the (really) big time. *Commun. ACM* **57**(6), 13–15 (2014). <https://doi.org/10.1145/2601069>
19. Rey, H.G., Pedreira, C., Quiñan Quiroga, R.: Past, present and future of spike sorting techniques. *Brain Res. Bull.* **119**, 106–117 (2015). <https://doi.org/10.1016/j.brainresbull.2015.04.007>
20. Richards, B.A., et al.: A deep learning framework for neuroscience. *Nat. Neurosci.* **22**(11), 1761–1770 (2019). <https://doi.org/10.1038/s41593-019-0520-2>
21. Smirnova, L., et al.: Organoid intelligence (OI): the new frontier in biocomputing and intelligence-in-a-dish. *Front. Sci.* **0** (2023). <https://doi.org/10.3389/fsci.2023.1017235>
22. Takahashi, K., et al.: Induction of pluripotent stem cells from adult human fibroblasts by defined factors. *Cell* **131**(5), 861–872 (2007). <https://doi.org/10.1016/j.cell.2007.11.019>

23. VanRullen, R., Guyonneau, R., Thorpe, S.J.: Spike times make sense. *Trends Neurosci.* **28**(1), 1–4 (2005). <https://doi.org/10.1016/j.tins.2004.10.010>
24. Zhang, X., et al.: ‘Mind in vitro’ platforms: versatile, scalable, robust and open solutions to interfacing with living neurons (2023). <https://doi.org/10.1101/2023.08.21.554033>
25. Zhao, Z., et al.: Organoids. *Nat. Rev. Methods Primers* **2**(1), 1–21 (2022). <https://doi.org/10.1038/s43586-022-00174-y>



# Verification of Concurrent Programs Using Hybrid Concrete-Symbolic Interpretation

Emily Tucker and Louis-Noël Pouchet<sup>(✉)</sup>

Colorado State University, Fort Collins, CO, USA  
{Emily.Tucker,Louis-Noel.Pouchet}@colostate.edu

**Abstract.** Source-level transformations of programs are fundamental to achieve high-performance: complex loop transformations, including loop tiling and parallelization, must typically be applied by a user or an automated tool. However this process is error-prone, especially when combined with transformations of the data layout, code structure and statements themselves.

In this work, we present an approach to prove the equivalence between a function and its candidate optimized version which is mostly agnostic to the schedule and storage implemented. It can prove the equivalence between a sequential function and its parallelized version, under practical restrictions.

**Keywords:** Verification · Program Equivalence · Concurrent Programs

## 1 Introduction

Optimizing compilers must provide a semantically equivalent implementation to the input program being compiled, while optimizing the program description to efficiently map it onto a target hardware. Typically, *parallelism* should be exposed, be it at the instruction level, using SIMD vectorization, or using coarser-grain multi-thread parallelization. This parallelism may be automatically exposed by the compiler, or be assisted by the programmer's rewriting and annotations of the program to specify how parallelism may be implemented. An upside is the potential performance of parallel programs versus their sequential implementation. A downside is the risk for the compilers and humans alike to be buggy: such parallelizing transformations may break the original program dependences, be incorrect due to possible race conditions or deadlocks, or simply not following the sequential program semantics due to mismatches between the operations computed in the original and transformed programs.

In this work, we present a verification approach to prove the correctness of a set of parallelizing transformations. It is based on a hybrid concrete-symbolic interpreter, to verify the correctness of a pair of programs at the source level [26]. These programs, expressed in C semantics, may contain OpenMP directives for parallelization. Our tool can prove the absence of race conditions and deadlocks

in each of them, as well as prove the semantic equivalence between them: that is, for any input, they both produce the exact same output, necessarily.

The problem of determining the absence of races or deadlocks in parallel programs has been studied from multiple angles ranging from static analyses e.g. [5, 7, 34, 38], dynamic analyses [3, 8] including intercepting the OpenMP runtime [15], symbolic analyses [26, 30, 32], as well as using Coq-formalized proofs [11]. Program equivalence itself has been studied from these angles, including for affine programs [4, 36], and by symbolic execution [17, 28, 32] and concrete-symbolic interpretation [26]. We develop an approach which relies on concrete interpretation of the control-flow instructions, therefore limited to a class of programs: those with *statically interpretable control-flow* (SICF) [26]. In turn, this enables a program equivalence approach that has linear time and space complexity with respect to the number of operations *executed* by the program, in a manner fully independent from the syntax used, schedule of operations, and storage implemented [26]. Extending to support parallel programs incurs a low-overhead additional complexity, and can prove for sequentially consistent programs the absence of parallelization errors, and full equivalence between a parallel and a sequential program. We make the following contributions:

- We outline how our hybrid concrete-symbolic interpreter proceeds to verify the correctness of some forms of concurrent programs.
- We introduce a translation of several OpenMP constructs into this framework, enabling the detection of race conditions for a class of OpenMP programs.
- We present experimental results demonstrating the ability of our system to catch concurrency bugs in programs generated by an auto-parallelizing compiler, and prove correct the parallelization implemented otherwise.

## 2 Background and Overview

We first summarize key aspects of the CDAG-based verification approach [26], before developing the ideas for the verification of a class of concurrent programs.

### 2.1 Hybrid Concrete-Symbolic Verification for SICF Programs

Our approach is based on *concrete* interpretation of selected instructions in the input program, while treating any other operation as *symbolic* and building a symbolic representation for these based on Computation Directed Acyclic Graph (CDAG) [13, 18, 24]. Specifically, any conditional branch instruction, and the operations transitively involved in computing the value of these condition(s), as well as operations involved in address calculations, need to be concretely interpretable from the input program. Any other operation can be symbolic: the concrete value of their operands need not be known at interpretation time. This leads to a class of statically interpretable control-flow programs we support: those where all branches to be taken by the program (i.e., the control-flow) when



executing on the target machine can be discovered by concrete interpretation of the relevant operations from the input program.

We remark a relation with Static Control-Flow Programs (SCoP) [14], known as affine programs [22, 27]: for these programs all branches to be taken can be exactly characterized by *static analysis* and modeled using affine relations. SICF programs do not require any affine structure, do not depend on how loops and control-flow are implemented, but does require to know the actual problem size (e.g., loop bound) of SCoPs. Parametric loop bounds are not supported as we require the concrete values for each conditional branch to be taken by the interpreter. All SCoPs with known values for the parameters are SICF.

Another important restriction of our approach is the requirement to map each memory cell that is live-in/live-out to a unique name, to be used to reason on the memory values of both programs. We therefore typically target the verification of a pair of functions  $f_{orig}, f_{opt}$  (themselves possibly calling other functions which are also interpreted) such that  $f_{opt}$  is an optimized version of  $f_{orig}$ , meant to replace it in a larger program. They would therefore inherit the same *execution context* necessarily, making the verification possible [12].

*Illustrative Example.* We present below a simple example of our CDAG-based concrete-symbolic interpretation. The interpreter is equipped with a *concrete evaluator* which implements exactly the same concrete semantics as the target machine, making simplification of expressions by concrete interpretation valid. Every operation that can be concretely interpreted is; otherwise, the operation is promoted to a symbolic CDAG representation. CDAGs are schedule-independent and storage-independent, and represent the ordered set of operations that produce a value as a function of only live-in symbolic values and constants [18, 26].

Equivalence is achieved by showing that for the same live-out memory cells of both programs, the CDAGs constructed for them for  $f_{orig}$  and  $f_{opt}$  are semantically equivalent. In its simplest form, they can be strictly identical (making this check linear time), or possibly some semantics-preserving rewrites of the CDAGs may be needed first, e.g. to support associativity and commutativity [26].

CDAGs are used to reason on I/O lower bounds, given they are agnostic to scheduling and storage implemented. They represent the fundamental nature of the computation, not how it is implemented [18]. However as seen below they can grow exponentially: the values used for the second matrix-multiply are themselves a CDAG. To control complexity and build a representation during interpretation that is linear in space wrt. the operations executed, we deploy memory pooling of distinct (sub-)CDAGs, and use pointers to them. For multiple references to the same memory cell, only a pointer to its CDAG is duplicated.

NI = NJ = NK = NL = 2

```

/* D := alpha*A*B*C + beta*D */
for (i1 = 0, pos = -1; i1 < NI; i1++)
  for (j = 0; j < NJ; j++) {
    buffer[++pos] = 0.0;
    for (k = 0; k < NK; ++k)
      buffer[pos] += alpha*A[i1][k]*B[k][j];
  }
for (i2 = 0; i2 < NI; i2++)
  for (j = 0; j < NL; j++) {
    D[i2][j] *= beta; pos = i2*NJ;
    for (k = 0; k < NJ; ++k)
      D[i2][j] += buffer[pos++] * C[k][j];
  }

```

Interpret the CFG: concretely compute values taken by i1,j,k etc., and treat everything else symbolically, including live-ins A[0][0], ..., D[1][1]

```

// after interpreting i1 loop in full:
buffer[0] := 0.0 + alpha * A[0][0] * B[0][0] +
           alpha * A[0][1] * B[1][0];
// after interpreting i2 loop in full:
D[0][0] = beta * D[0][0] +
          (0.0 + alpha * A[0][0] * B[0][0] +
           alpha * A[0][1] * B[1][0]) * C[0][0] +
          (0.0 + alpha * A[0][0] * B[0][1] +
           alpha * A[0][1] * B[1][1]) * C[1][0]
D[0][1] = beta * D[0][1] +
          (0.0 + alpha * A[0][0] * B[0][0] +
           alpha * A[0][1] * B[1][0]) * C[0][1] + ...
// CDAG with pooling:
D[0][0] := beta * D[0][0] + ctags[id1] * C[0][0] +
           ctags[id2] * C[1][0]

```

## 2.2 Extending to Support Concurrent Programs

Prior work generalized this sequential verification approach to handle limited forms of concurrency: specifically program regions executing concurrently, synchronized using blocking FIFOs [26]. This support enables the verification of source-to-source transformations for high-performance accelerator designs targeting High-Level Synthesis toolchains, enabling for instance the verification of correctness of a systolic array implemented over 140k LoCs, using a  $64 \times 64$  array of concurrent units [37] (8k Processing Elements in total, interconnected using 16k FIFOs) in about 15 min, using a single CPU core [26].

We build on this approach to enable a preliminary support of some OpenMP parallelization of C programs, as described in Sect. 3. In a nutshell, the verification of concurrent programs works as follows. (1) The interpreter is extended with support for concurrent regions, which can be scheduled for interpretation akin to a multiprogramming approach in operating systems, switching between concurrent regions ready to execute. It implements interrupts (e.g., when a blocking FIFO is not ready) and regions can be ready to execute, executing, waiting (on FIFO readiness) or terminated. (2) It maintains virtual timestamps for the operations interpreted, capturing the earliest (virtual) time at which blocks of operations can execute, and associate each shared memory accesses with such timestamps. (3) It maintains histories of the (shared) memory accesses, enabling a check for non-determinism in case of accesses to the same memory location at the same virtual timestamp: if two possible valid concurrent schedules lead to a different memory state, the interpreter aborts. We use these features to develop a verification approach for a set of C+OpenMP parallel programs.

## 3 Verifying a Set of Concurrent Programs

We now detail our approach to verifying the subset of concurrent programs supported in this work. Specifically, we target the verification of a pair of functions  $f_{orig}, f_{opt}$  such that one is a substitute for the other in a larger program, both expressed as source-level C programs, possibly containing a subset of OpenMP pragmas as described in Sect. 4. If successful, the verification approach proves

that both programs have the same semantics in that they both compute the same result, if given the same inputs; and that for any synchronization-preserving concurrent schedule following the OpenMP pragmas specification, no race condition nor deadlock can occur. As shown in Sect. 5, this enables the verification of the correctness of source-to-source parallelizing compilers such as Pluto and PoCC which rely on inserting `#pragma omp parallel for` around parallel affine loops, including whether variables are properly privatized, whether complex tiled wave-front parallelization was properly implemented, etc.

### 3.1 Detecting Non-determinism

A key aspect of our approach is to significantly limit the form of memory consistency we support. During interpretation, if two operations (from two different logical threads) can occur at the same timestamp, and they both access the same shared memory location, then neither of these two operations can be a write. Otherwise the result may be non-deterministic: depending on which of the two operations in practice would execute first, the state of memory may be different. We perform such detection for both concrete and symbolic memory locations.

If two accesses (one being a write) to the same shared memory location need to be performed, we require a synchronization (such as, but not limited to, a barrier) between them, forcing sequentially consistent behavior for these two accesses. In practice, the virtual timestamp we maintain to determine which operations may execute at the same time typically changes only when a synchronization is interpreted, as it ensures every operation depending on it is executed after operations leading to it.

### 3.2 Verified Properties for Concurrent Programs

To verify a class of OpenMP programs, we assign *every* block of code that may execute in parallel (as per the OpenMP pragmas) to a distinct *logical* thread, as discussed in Sect. 4. For example, an `omp for` loop with 1000 iterations can be represented with 1000 logical threads. Therefore all possible serializations of these logical threads (e.g., different OpenMP schedules and their mapping to physical threads) will only reduce the amount of parallelism considered, making the analysis conservative but safe.

To model synchronizations, we rely on our own API to handle interrupts/wait/resume for logical threads. Note we assume every operation within a thread executes serially, in the *source program* order. This facilitates debugging at the source level, however it does not model the effect of compiler optimizations in the binary eventually executed, which may include reordering of operations.

**Definition 1 (Virtual timestamp and synchronization).** *Given two operations  $o_1, o_2$  executed by two logical threads  $l_1, l_2$ . A virtual timestamp  $t(o)$  is assigned to every operation, so that  $t(o_1) < t(o_2)$  iff there exists a point-to-point synchronization operation  $o_s : l_1 \rightarrow l_2$ , and  $o_1$  executes before  $o_s$  in  $l_1$ 's sequential order, and  $o_2$  executes after  $o_s$  in  $l_2$ 's.*

This relation amounts to the happens-before relation [15], that is, the necessity for all operations that execute serially before a point-to-point synchronization to all be terminated before the start of operations executing after this synchronization. Consequently, operations may have the same virtual timestamp, if they are not (transitively) ordered via synchronizations. Such operations may therefore execute at the same time, and we check whether they can produce a race by tracking the timestamp of operations accessing all non-thread-local data. We operate with a zero-latency model, that is we assume all operations execute in zero cycles, and only synchronizations can force a (partial) order between operations in different logical threads.

We detect the existence of a possible race condition conservatively, in this zero-latency model, by determining if there is any memory location accessible by two or more logical threads at the same time.

**Definition 2 (Read-write conflict).** *Given two operations  $o_1, o_2$  executed by two logical threads  $l_1, l_2$  such that  $t(o_1) = t(o_2)$ , which both access the same shared (non-local) memory location, and one of these accesses is a write. Then the operations expose a read-write conflict.*

It is also possible in general to create a deadlock situation, where point-to-point synchronizations are incorrectly implemented leading to a blocking state, that is, one or more logical thread cannot terminate.

**Definition 3 (Deadlock).** *Given a logical thread  $l$  in a blocking/interrupted state. If there is no other logical thread that can become active by interpretation, and which can modify the semaphore(s) on which  $l$  is blocked, then  $l$  is deadlocked.*

For our system to output a conclusive analysis of a program, it must reach termination without error. Without loss of generality, we assume every program is encapsulated in a function, and a single exit point exists for this function.

*Property 1 (Termination).* If the concrete interpretation of the control-flow and dataflow addressing reaches the main function's exit point, no read-write conflict was detected, and no deadlock is detected, then the interpretation terminated.

Note we implement an inelegant yet practical approach to handle non-terminating programs (e.g. infinite loops): a counter of the number of operations interpreted. If a maximum limit threshold is reached, the interpretation aborts.

We conclude with the equivalence between a pair of programs  $f_{orig}, f_{opt}$ .

*Property 2 (Equivalence between programs).* Given  $f_{orig}, f_{opt}$  two functions such that one is a substitute for the other in a main program, and all their arguments are non-aliasing and referencing all destinations of side-effects. If their interpretation terminates, and for every live-out memory location referenced in their arguments, the concrete values or CDAGs produced for the same location are semantically equivalent, then the two functions are equivalent.

A proof of an equivalent version of this property is outlined in [26].

### 3.3 High-Level Verification Procedure

We summarize the verification process as follows.  $f_{orig}, f_{opt}$  are each independently interpreted, computing for each a final memory state  $M_f$ . For each function  $f$ , the interpreter proceeds as described in prior work [26], interpreting the code for a thread until termination, or interruption due to a blocking synchronization primitive. A scheduler context-switches to the next thread in the queue in case of interrupt, until there is no more interrupted nor active thread in the queue (otherwise a deadlock is occurring). During interpretation, we associate a thread-specific timestamp to every non-local memory element read/written by each thread, itself only increased when synchronizations are interpreted. If a location has been written by a thread, no other thread can read-write it at the same timestamp, otherwise a read-write conflict is produced. We conservatively assume pointers to data initialized prior to the start of a concurrent region are shared between threads in this region. Pointer arithmetic besides thread-local declarations must lead to addressing these shared pointed locations, otherwise the interpreter aborts. As we operate at the source-level, scalar and array variables locally declared within a thread, and their scope of liveness, is trivially detected. Consequently, every variable that is not accessible at the end of the function is deleted from  $M_f$ . We do not support external functions (e.g., `libc`).

If interpretation terminated, we obtained  $M_{f_{orig}}$  and  $M_{f_{opt}}$  two memory states made only of variables accessible when the function terminates: live-in and live-out data [26]. As we restrict to programs where a unique name can be built for every memory cell, (e.g.,  $A[42][51]$ , `foobar`, etc.), requiring a lack of aliasing on the main function parameters, we simply proceed by checking for every such name that the memory values computed (be they concrete, or symbolic CDAGs) are semantically equivalent, e.g.  $M_{f_{orig}}(A[42][51]) \equiv M_{f_{opt}}(A[42][51])$ . Note we support a variety of rewrite rules on CDAGs prior to checking equivalence, e.g. to normalize associative/commutative reductions [26].

## 4 OpenMP

### 4.1 API For Concurrent Programs

We now briefly present our API for interpreting concurrent programs, and the translation of some OpenMP constructs with it.

At its core, only three API functions are needed. First we declare blocks of code to execute concurrently within a parallel region `regionid`, using the `register_concurrent(regionid, block)` for each such block. When interpretation of the code block assigned to a thread reaches the end of its control-flow, the thread is terminated. Note there is an infinite number of logical threads available. Second, we declare a point-to-point synchronization with a semaphore, using `set_semaphore_value(regionid, semid, value)` and the associated blocking function `wait_until_semaphore_value(regionid, semid, value)`. These calls are inserted in the program as regular C function calls, and different threads

may read/write the same semaphores. They form a sufficient flexible set of constructs (fork/join, and point-to-point synchronization) to emulate the OpenMP constructs we target in this work.

## 4.2 OpenMP Constructs

**#pragma omp parallel** A parallel OpenMP region, outside of any **for** loop, requires knowledge of the concrete number of threads to be verified. The code block dominated by the pragma is recorded for parallel execution, with one call to **register\_concurrent(regionid,block)** per thread. Then, the block in the main program is replaced by a pair of calls to be executed by the main/-master thread: **concurrent\_region\_start(regionid)** and the associated join **concurrent\_region\_end(regionid)**, which emulates the fork/join OpenMP model. When interpreting the **start** call, the interpreter creates the requested threads, and executes them in order, context-switching to the next thread only if the current thread has terminated, or is interrupted due to a blocking call (e.g., waiting on a semaphore to reach a particular value). The **end** call acts as a global barrier for the region, except the threads are deleted, this only once all have reached the end of their control-flow. Otherwise a deadlock is detected.

**#pragma omp for** By design the interpreter requires loop bounds to be concretely interpretable, from the statically-interpretable control-flow requirement. Consequently, the loop bound expressions are known and the code blocks for the concurrent region can be seamlessly updated with the set of iterations to be run by each thread, the code **block** becoming the loop body. We support the **static** schedule, as well as a conservative mode where the number of logical threads is dynamically increased to assign one loop iteration per thread. **private** and related clauses are translated to equivalent local variable declarations and initializations explicitly by pre-processing. A **barrier** is automatically inserted at the loop exit, unless the **nowait** clause is specified.

**#pragma omp section** OpenMP sections are translated similarly to parallel regions, with the proper code blocks registered for concurrent execution.

**#pragma omp barrier** Finally, a barrier is implemented via semaphores, similar to the point-to-point synchronization described above. Technically, we use a single collective synchronization API **wait\_on\_semaphores(regionid,sem\_array)** interpreted by all threads, with each thread writing its own semaphore when it reached the API call, and interpretation the next instruction after the barrier (if any) being implemented only when all threads wrote their semaphore, to avoid ordering issues.

## 4.3 Extensions and Ongoing Work

In this work we limit our presentation to the simplest form of **doall** and **doacross** parallelism using OpenMP, for illustration purposes. Our ongoing work includes support for a large subset of the OpenMP 4 constructs, including tasks and their

dependencies. Clearly, our flexible model for declaring concurrent regions and their synchronizations leads to easily “software-emulate” the various OpenMP constructs, by translating them to C code (e.g. to properly declare private variables, but also scheduling strategies) beforehand. As downside, this translation of pragmas we perform as pre-processing must itself be correct, and matching what the OpenMP-capable compiler implements when translating these pragmas.

Ongoing work also includes support for a subset of other parallel languages such as Habanero [6] and its C variant, essentially implementing a hybrid concrete-symbolic interpretation of program to compute the happens-before and may-happen-in-parallel relations in a manner agnostic to the syntax used to implement the program. Approaches using automated theorem proving have been developed [10], we aim for increased generality to how the code is implemented. We will discuss this generalized support during our presentation, and its trade-offs versus other verification approaches. We also mention the issue of properly verifying runtime systems in charge of orchestrating such parallel computations. While they can be simply made part of the programs to interpret, it is often desirable to instead perform a slightly more complex interpretation of the program (e.g., using infinitely many logical threads to decompose maximally the concurrency available) to make the verification robust to a variety of concurrent schedules that can be implemented by these runtimes, including work-stealing.

## 5 Experimental Results

### 5.1 Experimental Setup

For brevity we limit to illustrating the 2 mm benchmark from PolyBench/C [27], and will present extensive results over all PolyBenches during our talk [26]. 2mm computes  $\beta * D + \alpha * A * B * C$  for rectangular matrices  $A, B, C, D$ : the product of three matrices. The matrix data type is symbolic (and hence the verification holds for “any” data type to be used), and we vary the problem sizes and program transformations (including OpenMP parallelization) computed by the PoCC compiler [2]. All experiments are conducted on a single core of an AMD Ryzen 5900HX, using 64 GB of DDR4 RAM, all optimizations within the interpreter are disabled except storing duplicate sub-CDAGs by pointers.

## 6 Results on 2mm benchmark

Table 1 displays the time to Interpret two programs: a sequential, base version and a transformed version optimized by PoCC using fusion, tiling and OpenMP parallelization. We display the number of statement instances in the source code interpreted, and the time to compute equivalence (note the time to detect read-write conflicts, if any, is integrated in the interpretation time), the number of CDAG nodes created, and the maximal memory usage during the full process. All experiments model 8 logical threads. Time is barely sensitive to higher threads count, but it is influenced by the number of non-barrier synchronizations.

**Table 1.** Summary of experiments on PolyBench/2mm benchmark

Benchmark	Int. seq.	#stmts	Int. PoCC	#stmts	Equiv.	#nodes	Max Mem
2 mm-128	30.4 s	12 M	32.8 s	12.7 M	2.6 s	4.2 M	3 GB
2 mm-200	106 s	43 M	116 s	45 M	10.3 s	14 M	10 GB
2 mm-32	0.50 s	207k	0.54 s	207k	0.04 s	67k	53 MB
2 mm-32-ns	0.80 s	207k	0.84 s	207k	0.47 s	67k	938 MB
2 mm-200-np	106 s	43 M	116 s	45 M	N/A	14 M	10 GB
2 mm-200-bt	106 s	43 M	113 s	44 M	10.0 s	14 M	10 GB

Table 1 reports two cases of equivalence for square problem size 128, and the rectangular MEDIUM polybench size (−200). Throughput is around 0.4 M statements per second. For 2 mm, memory use can exceed 50 GB for problem size of 512 [26], showing a scalability limit with this approach. Solutions to overcome this limit includes on-the-fly CDAG compression by affine folding [29], allowing interpretation to scale gracefully to “arbitrary” problem sizes. We also display the benefits of avoiding duplication of CDAG subtrees, with the -ns variant for  $N = 32$  disabling this optimization. For this small problem size, memory usage is already 20x larger without this optimization, and would prevent scaling to more realistic problem sizes. We also report two bugs we manually introduced: a missing inner loop iterator in the private clause (-np) and a loop bound error in the tiling, leading to skip iterations (-bt). For both, the interpreter ran to completion to maximize the number of errors found.

## 7 Related Work

The detection of concurrency bugs and equivalence between two implementations are often split in two different problems. Detecting bugs in OpenMP programs has been effectively implemented [15], and various static analyses have also been proposed, e.g. [7, 38]. However none verifies at the same time the compliance of the parallelized program with the semantics of the *original* unoptimized program.

Our framework can prove the equivalence of C-style programs under a wide set of code transformations, albeit limited to the class of Statically Interpretable Control-Flow. An approach complementary to ours is KLEE [1, 28, 30]. KLEE implements a different symbolic interpretation approach; ours is specialized for equivalence of programs with a *single* concretely interpretable CFG path and concretely interpretable array subscripts, trading off generality for speed. We limit coverage to fixed problem sizes but can operate on a variety of symbolic data types, at order(s) of magnitude faster speed than KLEE(-float [20]).

Other approaches to verifying or guaranteeing the correctness of (parallel) programs include model checking e.g. [17, 21, 33, 35], translation validation e.g. [9, 23, 25] and of course using a certified compiler e.g. [16, 19, 31].



## 8 Conclusion

Although numerous approaches exist to assess the correctness of a program optimization, they are often associated with fundamental limitations to how the program is implemented. By using a hybrid concrete-symbolic interpretation approach, and imposing sensible restrictions to support only programs with statically interpretable control-flow, it is possible to achieve significantly stronger guarantees than testing, while offering full flexibility for *how* the program is implemented: arbitrary statement transformations, schedule (including parallel ones), arbitrary storage and bufferization schemes, etc. In this work we outlined a verification approach for (concurrent) programs based on prior work [26], and simple extensions to handle OpenMP programs and verify their equivalence. Future work includes generalizing to a large subset of OpenMP 4, and improving scalability further using a nearly constant-space approach to encode CDAGs.

## References

1. The KLEE symbolic execution engine (2023). <https://klee.github.io>
2. PoCC, the Polyhedral Compiler Collection 1.6 (2023). <https://pocc.sourceforge.net>
3. Atzeni, S., et al.: ARCHER: effectively spotting data races in large OpenMP applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 53–62 (2016)
4. Bao, W., Krishnamoorthy, S., Pouchet, L.N., Rastello, F., Sadayappan, P.: Polycheck: dynamic verification of iteration space transformations on affine programs. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2016)
5. Bora, U., Das, S., Kukreja, P., Joshi, S., Upadrasta, R., Rajopadhye, S.: LLOV: a fast static data-race checker for OpenMP programs. *ACM Trans. Archit. Code Optim.* **17**(4), 1–26 (2020)
6. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-java: the new adventures of old x10. In: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java. pp. 51–61 (2011)
7. Chatarasi, P., Shirako, J., Kong, M., Sarkar, V.: An extended polyhedral model for SPMD programs and its use in static data race detection. In: Languages and Compilers for Parallel Computing: 29th International Workshop, LCPC 2016, Rochester, NY, USA, September 28–30, 2016, Revised Papers 29, pp. 106–120. Springer (2017)
8. Cheng, G.I., Feng, M., Leiserson, C.E., Randall, K.H., Stark, A.F.: Detecting data races in CILK programs that use locks. In: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 298–309 (1998)
9. Clément, B., Cohen, A.: End-to-end translation validation for the halide language. *Proc. ACM Program. Lang.* **6**(OOPSLA1), 1–30 (2022)
10. Cogumbreiro, T., Shirako, J., Sarkar, V.: Formalization of habanero phasers using coq. *J. Logic. Algebraic Methods Program.* **90**, 50–60 (2017)
11. Cogumbreiro, T., Surendran, R., Martins, F., Sarkar, V., Vasconcelos, V.T., Grossman, M.: Deadlock avoidance in parallel programs with futures: why parallel tasks should not wait for strangers. *Proc. ACM Program. Lang.* **1**(OOPSLA), 1–26 (2017)

12. Cousot, P.: Abstract interpretation based formal methods and future challenges. In: *Informatics: 10 Years Back, 10 Years Ahead*, pp. 138–156. Springer (2001)
13. Elango, V., Rastello, F., Pouchet, L.N., Ramanujam, J., Sadayappan, P.: On characterizing the data access complexity of programs. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2015)
14. Feautrier, P.: Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *Int. J. Parallel Program.* **21**, 389–420 (1992)
15. Gu, Y., Mellor-Crummey, J.: Dynamic data race detection for openMP programs. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 767–778. IEEE (2018)
16. Herklotz, Y., Pollard, J.D., Ramanathan, N., Wickerson, J.: Formal verification of high-level synthesis. *PProc. ACM Program. Lang.* **5**(OOPSLA), 1–30 (2021)
17. Jakobs, M.C.: PEQCHECK: localized and context-aware checking of functional equivalence. In: *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormalISE)*, pp. 130–140 (2021)
18. Jia-Wei, H., Kung, H.T.: I/o complexity: the red-blue pebble game. In: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, pp. 326–333 (1981)
19. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert-a formally verified optimizing compiler. In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress* (2016)
20. Liew, D., Schemmel, D., Cadar, C., Donaldson, A.F., Zahl, R., Wehrle, K.: Floating-point symbolic execution: a case study in n-version programming. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 601–612. IEEE (2017)
21. Mercer, E.G., Anderson, P., Vrvilo, N., Sarkar, V.: Model checking task parallel programs using gradual permissions (n). In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 535–540. IEEE (2015)
22. Moses, W.S., Chelini, L., Zhao, R., Zinenko, O.: Polygeist: raising c to polyhedral MLIR. In: *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 45–59. IEEE (2021)
23. Necula, G.C.: Translation validation for an optimizing compiler. In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp. 83–94 (2000)
24. Olivry, A., Langou, J., Pouchet, L.N., Sadayappan, P., Rastello, F.: Automated derivation of parametric data movement lower bounds for affine programs. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2020)
25. Parthasarathy, G., Dardinier, T., Bonneau, B., Müller, P., Summers, A.J.: Towards trustworthy automated program verifiers: formally validating translations into an intermediate verification language. *Proc. ACM Program. Lang.* **8**(PLDI), 1510–1534 (2024)
26. Pouchet, L.N., et al.: Formal verification of source-to-source transformations for HLS. In: *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 97–107 (2024)
27. Pouchet, L.N., Yuki, T.: Polybench/c 4.2.1 (2023). <https://polybench.sourceforge.net>
28. Ramos, D.A., Engler, D.: {Under-Constrained} symbolic execution: correctness checking for real code. In: *24th USENIX Security Symposium (USENIX Security 15)*, pp. 49–64 (2015)

29. Rodríguez, G., Pouchet, L.N., Touriño, J.: Representing integer sequences using piecewise-affine loops. *Mathematics* **9**(19), 2368 (2021)
30. Schemmel, D., Büning, J., Rodríguez, C., Laprell, D., Wehrle, K.: Symbolic partial-order execution for testing multi-threaded programs. In: *International Conference on Computer Aided Verification*, pp. 376–400. Springer (2020)
31. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Comperttso: a verified compiler for relaxed-memory concurrency. *J. ACM (JACM)* **60**(3), 1–50 (2013)
32. Siegel, S.F., et al.: CIVL: the concurrency intermediate verification language. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12. ACM, Austin Texas (2015)
33. Siegel, S.F., et al.: CIVL: the concurrency intermediate verification language. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12 (2015)
34. Swain, B., Li, Y., Liu, P., Laguna, I., Georgakoudis, G., Huang, J.: OMPRacer: a scalable and precise static race detector for OpenMP programs. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14 (Nov 2020)
35. Vechev, M., Yahav, E., Raman, R., Sarkar, V.: Automatic verification of determinism for structured parallel programs. In: Cousot, R., Martel, M. (eds.) *SAS 2010*. LNCS, vol. 6337, pp. 455–471. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-15769-1\\_28](https://doi.org/10.1007/978-3-642-15769-1_28)
36. Verdoolaege, S., Janssens, G., Bruynooghe, M.: Equivalence checking of static affine programs using widening to handle recurrences. In: *Computer Aided Verification* (2009)
37. Wang, J., Guo, L., Cong, J.: AutoSA: a polyhedral compiler for high-performance systolic arrays on FPGA. In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 93–104 (2021)
38. Ye, F., Schordan, M., Liao, C., Lin, P.H., Karlin, I., Sarkar, V.: Using polyhedral analysis to verify openMP applications are data race free. In: *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)*, pp. 42–50. IEEE (2018)



# Scalable Small Message Aggregation on Modern Interconnects

Aaron Welch<sup>1</sup>(✉) , Oscar Hernandez<sup>1</sup> , Stephen Poole<sup>2</sup> ,  
and Wendy Poole<sup>2</sup>

<sup>1</sup> Oak Ridge National Laboratory, Oak Ridge, TN, USA  
{welchda,oscar}@ornl.gov

<sup>2</sup> Los Alamos National Laboratory, Los Alamos, NM, USA  
{swpoole,wkpoole}@lanl.gov

**Abstract.** The partitioned global address space (PGAS) model is popular for applying a classic shared memory approach to large systems, but some classes of problems rely on large numbers of small remote memory accesses targeting random locations across the network. On modern interconnects this can overwhelm the network, leading to message rate inefficiencies. This small message problem can be solved through aggregation strategies, however these typically require undesirable code restructuring that is cumbersome to incorporate and maintain in user applications. A strategy called “aggregation contexts” aimed at alleviating this burden has previously been proposed for the OpenSHMEM PGAS API. Despite its potential, it has not yet been validated for scalability on large systems consisting of thousands of nodes, nor proven to be performance-portable, which are critical for its adoption. In this paper, we demonstrate the scalability and performance portability of aggregation contexts using up to 8192 nodes on ORNL’s Frontier system. Our study reveals good scaling patterns while also identifying further opportunities for performance improvements to make it even more effective.

**Keywords:** OpenSHMEM · message aggregation · aggregation contexts · conveyors · many-to-many communication patterns

## 1 Introduction

Applications with many-to-many and irregular access patterns are prone to generating large amounts of small messages that tend to congest modern high-speed networks, limiting their performance. This small message problem can be solved through message aggregation, however most approaches tend to involve significant and undesirable code restructuring to use them. Previous work introduced an extension called “aggregation contexts” [9] to the OpenSHMEM [7] PGAS programming model that worked by implicitly deferring completion of operations performed on them, organised into work queues it would later send in bulk for local processing. In addition, since it was applied as an abstract header layered over the OpenSHMEM API, the strategy it implemented could easily be

adapted to other communication interfaces like MPI-3. The bulk of its reference implementation was achieved through the use of conveyors [4], an aggregation library from the bale effort, and thus its performance is also tied to this library.

While use of aggregation contexts results in changes to the synchronisation semantics of OpenSHMEM, they allow for significant performance improvements over independent atomic, get, and put (AGP) network operations by up to 65x, while largely preserving the application’s algorithmic intent. However, it has never been evaluated on any large-scale HPC systems with thousands of nodes. Such validation is crucial to ensure it is a scalable and performance portable solution that can be put to wider use and effectively map to multiple interconnect technologies, and is the primary goal of this paper. To this end, we tested aggregation contexts on ORNL’s Frontier (a TOP500 system), to determine whether their performance benefits persist with increasingly larger process/node counts and compare its performance to that of ORNL’s Andes and the previously tested HPC Advisory Council’s Iris, presented in Sect. 4.

The Frontier system is of particular interest due to its use of the Slinghost 11 network interconnect [1], which is relatively new and untested for such aggregation strategies. Testing how these capabilities affect message rates and the performance of small message aggregation is imperative. Since performance portability is also important to us, we compare equivalent executions across several different systems in Sect. 4.1 and discuss key observations of interest.

Section 3 provides a brief background on the design of conveyors and aggregation contexts so as to be able to reason about how these design choices can affect the observed patterns. After our initial scalability study on Frontier, we will further discuss the behaviours observed and speculate on potential opportunities to improve upon them in Sect. 5. Finally, we will provide our conclusions in Sect. 6 along with our expectations for what our findings may mean for how our aggregation strategy may translate to other and future systems.

## 2 Related Work

Slingshot is a novel interconnect designed to optimise network operations through its Rosetta switch, which implements adaptive routing, congestion control, and quality of service (QoS). The capabilities of this Slingshot interconnect system have been evaluated in the context of Slingshot 10 and 11. De Sensi [1] focused on evaluating how Slingshot 10 mitigated the impact of congestion on network latency and bandwidth in benchmarks and mini-apps compared to the Aries interconnect. Khorassani [3] evaluated the scalability of MPI implementations on Slingshot 10, using Cray MPICH, Open MPI/UCX/RCCL, and MVAPICH2 on both CPU and GPU, as well as a preliminary study of CPU-only on Slingshot 11. Namashivayam [5] focused on the early implementation of OpenSHMEM-X on Slingshot 11 NICs and proposed extensions to enhance performance. The study provides an overview of the supported features of the Slingshot 11 NIC, along with high-level implementation details and a detailed performance analysis using microbenchmarks and application kernels. However,

none of these studies evaluated the small message rate problem on Slingshot 11 at the scale of this paper.

Venkatesan [8] implemented an OpenSHMEM extension for aggregation called queues that takes advantage of QoS on Infiniband interconnects. The extension exposes queue operations with semantics similar to conveyors, but it requires significant restructuring of applications similarly to conveyors and was not evaluated at scale. Paul [6] proposed an actor model within PGAS for message aggregation that resulted in significant performance improvements. Similarly, CAL [2] offers more comprehensive high-level abstractions for aggregation in the Chapel programming language, including support for maps, scans, and reductions. These approaches would require porting the OpenSHMEM AGP style of code to the actor model or aggregator objects in Chapel applications, thereby increasing the effort for the user.

### 3 Background

Before we can examine our initial results, we must first describe the internal workings of both conveyors and aggregation contexts. More details on each can be found in [4] and [9], respectively.

#### 3.1 Conveyors

Conveyors are an abstraction around message queues, allowing a process to *push* items that will eventually be *pulled* from a given target process. The primary communication unit for conveyors is user-specified fixed-size items packed into similarly fixed size incoming/outgoing buffers that can be transmitted more coarsely. Much of its management is kept hidden, though the actual processing of received message items is provided by its user.

During their primary operation, users are allowed to do one of four actions—*push/pull* encoded items, *unpull* the last pulled item, or *advance* the conveyor in order to help ensure progress. This produces a clear pattern for the general use of conveyors—push messages to target processes until doing so fails due to lack of space to enqueue, then pulling all incoming messages to process them until there are no more that can be acted upon, and finally wrapping the previous two phases within a loop that advances it. There are no guarantees provided for precisely when individual messages are sent, which will typically happen as outgoing buffers fill up, even without a direct call to advance, provided there is currently space on the receiving end.

#### 3.2 Aggregation Contexts

Aggregation contexts were implemented on top of conveyors in order to abstract their execution and associated requirements away from the user of an OpenSHMEM application. Crucially, they were designed so as to require minimal change to applications and retain the original semantic meaning of their communication

operations, trading potential further performance gains with ease of development/maintenance. What this entailed was creating an abstract context object specifying that aggregation is desired, which then implicitly alters the completion semantics of all future operations performed on that context, loosening them as much as possible so that no local or remote completion is guaranteed until the user requires it by flushing all pending communication via a *quiet*. After such quiet operations, contexts remain continuously available for future operations with no explicit state management.

Contexts are able to exploit the fact that OpenSHMEM has a limited set of communication operations it can perform, and thus it just needs to provide a generic progress loop that is able to recognise and act upon any of the possible network operations. Thus, contexts need to specify a fixed-size packet structure capable of encoding any of these operations so that the same conveyors/buffers can be used across all of them, which at a minimum required the specification of an operation type, local and remote addresses, and value. Since not all operations would require each field, this results in efficiency concerns that direct use of conveyors would not have, particularly with the likes of atomic increments such as those employed by bale’s histogram, which only require one field (the remote address) beyond the type. This was addressed by adding a second level of deferment to pack up to three operations for each such increment or similar, and sending them through conveyors as a single unit.

## 4 Empirical Study

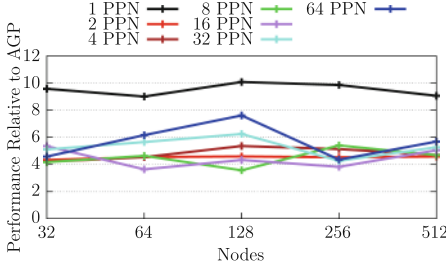
For our scalability studies, we used the histogram application from bale classic<sup>1</sup>. Histogram exhibits a communication pattern in which each processing elements PEs asynchronously send many independent updates to a distributed table in a random many-to-many communication pattern. In effect, this is an attempt to perform data binning on a distributed shared data set. The application simulates this by generating a uniform list of random table indices during an initialisation phase and then timing how long it takes to atomically increment the value at each random index. This is the simplest bale application, but it represents a common communication pattern and is often used to build more complex patterns, including in other bale applications (e.g., sparse matrix transpose). It also provides the most direct method of testing the performance features of conveyors and aggregation contexts.

We used three versions of histogram—the first uses OpenSHMEM AGP operations directly, while the other two use conveyors or aggregation contexts. We are interested in two metrics: the raw message rate of the conveyor and context implementations in terms of remote messages/updates per second per core/PE, and the relative speedup they provide over the AGP version (e.g.,  $\frac{\text{perf}_{\text{conveyors}}}{\text{perf}_{\text{AGP}}}$ ).

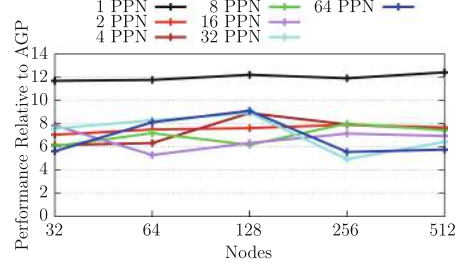
The following testing was performed on 32–512 nodes of ORNL’s Frontier supercomputer, using 1–64 processes per node (PPN) with a block distribution.

<sup>1</sup> [https://github.com/jdevinney/bale/tree/master/src/bale\\_classic](https://github.com/jdevinney/bale/tree/master/src/bale_classic).

We used Cray clang 15.0.0 and Cray OpenSHMEM-X 11.7.2.3 with XPMEM for building and running histogram over the Slingshot 11 interconnect. Additionally, we used the SLURM hint “-hint=nomultithread” to map one PE per core. We also extended this same testing to as many as 8192 nodes using an older version of OpenSHMEM-X in Sect. 5.

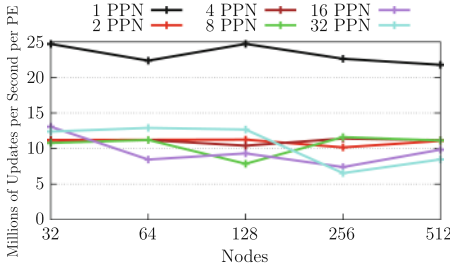


**Fig. 1.** Speedup Scaling for Conveyors.

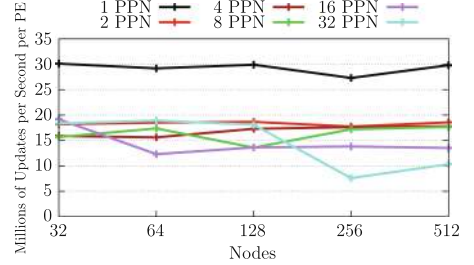


**Fig. 2.** Speedup Scaling for Contexts.

Figures 1 and 2 show the relative speedup over AGP for the conveyor and context implementations, respectively. It can be seen that these aggregation strategies compare well against AGP, even through to high node and PE counts, with stable, flat lines, demonstrating good scaling. Additionally, despite aggregation contexts being implemented on top of conveyors, it can actually be seen edging out over them in relative performance, easily achieving gains of about 40%. This is the result of the packet buffer optimisation described in Sect. 3.2, reducing the number of required calls into the conveyor API by 33% while sending the same number of messages.



**Fig. 3.** Message Rate Scaling for Conveyors.



**Fig. 4.** Message Rate Scaling for Contexts.

Although we are focused on comparing our performance to AGP, we are also interested in understanding the message rate that these strategies can achieve independently. Figures 3 and 4 show the absolute message rate per PE for the conveyor and context implementations. Since these tests demonstrate weak scaling, the continued persistence of flat scaling lines proves that not only does conveyor’s aggregation perform well with respect to classic AGP operations, but that the resulting raw performance also improves nicely as a factor of the pro-



cess count, with double the processes equating to roughly double the aggregate message rate.

A key observation is that 1 PPN outperforms 2–64 PPN by a significant factor. This is not unexpected, as with a high PPN, conveyors must contend with shared resources to send local and remote buffers. Another important question is just how efficient is the performance we are scaling of conveyors and aggregation contexts and if it is performance-portable across platforms. We discuss the first question in Sect. 5 and the latter in Sect. 4.1.

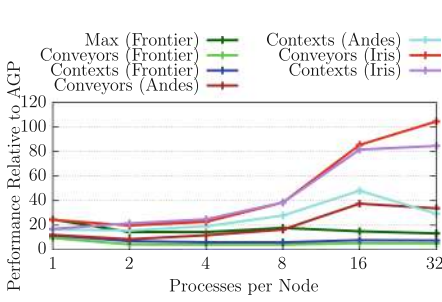
#### 4.1 Performance Portability

To evaluate the performance portability of conveyors and aggregation contexts across different systems and network architectures, we tested them using histogram on 32 nodes with 1–32 PPN, on each of the HPC Advisory Council’s Iris and ORNL’s Andes and Frontier systems. The goal was to evaluate their performance using common runtime configurations.

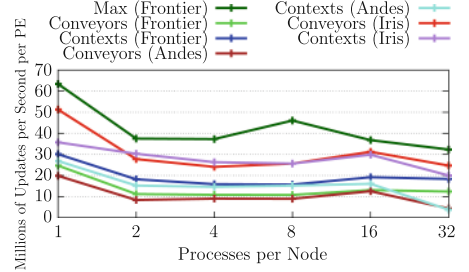
We also include results for a modified version of the aggregation context implementation to add an additional point of comparison in an attempt to project the maximum possible performance that we could potentially achieve with our current aggregation strategy. This modification simulated the strategy used in aggregation contexts, but without conveyors or any other overhead incurred by waiting for anything to complete while still performing all the work that contexts would ordinarily require (i.e., memory updates). That is, if an update is destined for a PE on the same node, it updates the equivalent index of its own memory (so as to avoid the simulation introducing cache competition), whereas if it is for another node it packs it the same way it usually would, but upon sending processes the messages itself.

This provides a good approximation for the amount of work “pulling” data that a PE would need to perform due to the uniform distribution of updates in histogram. It continues in this manner until all updates are complete. While this strategy invalidates the correctness of the results, it allows us to simulate the performance of an unrealistically ideal scenario with perfect progress, which we simply termed “Max” and ran alongside conveyors and contexts on Frontier for comparison.

Frontier’s environment was set up the same as in Sect. 4, although Iris and Andes used OpenMPI 5.0.3 for its implementation of OpenSHMEM 1.4, built with Unified Communication X (UCX) 1.16.0 also using XPMEM. Iris is a Dell C6400 32-node cluster with dual socket Intel Xeon 8280 CPUs, 192 GB of 2666 MHz DDR4 memory, and NVIDIA ConnectX-6 HDR100 100Gbit/s InfiniBand/VPI NICs connected via an NVIDIA HDR Quantum QM7800 switch. Andes is a 704-node cluster whose nodes contain two 16-core 3.0GHz AMD EPYC 7302 processors, 256GB of main memory and an NVIDIA ConnectX-6 HDR200 InfiniBand NIC.



**Fig. 5.** Speedup Comparison, 32 Nodes.



**Fig. 6.** Message Rate Comparison, 32 Nodes.

Figure 5 might initially suggest declining performance from Iris to Frontier, but looking at Fig. 6 tells a different story. While there are some performance differences between systems regarding updates per second per PE for conveyors and aggregation contexts, the most notable change is the narrowing gap between them and AGP performance, particularly on Frontier’s Slingshot 11 interconnect. This is good news, as the AGP version is the preferred programming model approach because of its simplicity and will remain widely in use within existing code bases. In all cases, there are still substantial improvements to be gained by using aggregation, though the cost for not doing so may not be as drastic on newer network interconnects as they improve their message rate performance.

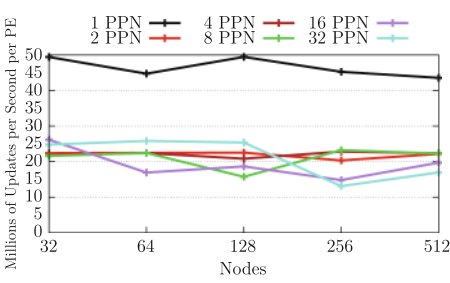
On the other hand, the benefits of using aggregation continue to be significant. The use of these aggregation strategies not only saves resources but also effectively mitigates platform differences, demonstrating consistent performance portability across different systems. Furthermore, the comparison to our simulated maximum performance on Frontier shows that our results are remarkably close to the best possible per-process performance we could achieve with our current hardware and aggregation strategies on Frontier. However, with the message rate scaling remaining flat across architectures and through up to 64 PPN, this suggests that aggregate message rate may continue to predictably rise with more cores up to some unknown value, potentially even if those cores are weaker.

## 5 Future Optimisation Opportunities

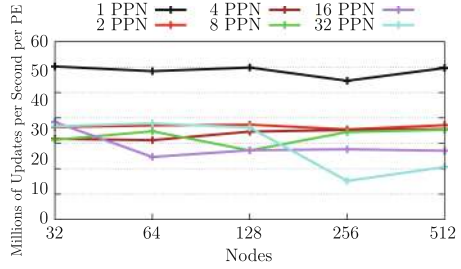
In this section, we describe some remaining areas of investigation with potential performance impacts which may contain opportunities to further improve the efficiency of message aggregation in conveyors and aggregation contexts.

### 5.1 System Scheduling Impact

The following observations come from the fact that we ran many of our tests at two different points in time with regard to Frontier’s overall workloads. While the latest of the results was presented in Sect. 4 in order to make use of a newer OpenSHMEM-X version, we had prior runs using version 11.5.7 that are also noteworthy, seen in Figs. 7 and 8.



**Fig. 7.** Message Rate Scaling for Conveyors (Exclusive Access).



**Fig. 8.** Message Rate Scaling for Contexts (Exclusive Access).

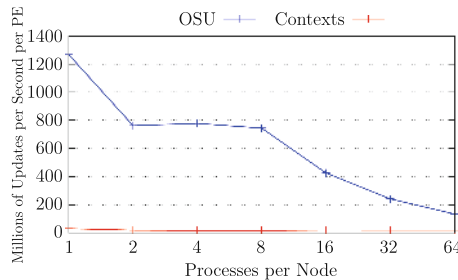
First we note was that during our initial access to Frontier, we were also able to run all histogram implementations all the way up to 8192 nodes and still maintain our same scaling behaviour, proving the scalability of aggregation beyond any doubt. The latest testing did not show any appreciable performance differences between the two OpenSHMEM implementation versions, however the results we saw for equivalent runs was dramatically different than our prior findings by roughly 2x. Crucially, the AGP performance was largely unaffected, meaning the performance gap between it and the aggregated approaches narrowed further still. The only difference was that our prior runs were with exclusive access to Frontier, whereas the latest ones were under normal operating workloads, sharing system resources such as the network interconnect with other users. This implies that conveyors may be more sensitive to shared network resources and that capabilities such as Slingshot 11 congestion control are important performance considerations, though this can be hard to control for and thus we are unable to get conclusive data at this time.

## 5.2 Aggregation Efficiency

Next, we would like to determine how efficiently these aggregation strategies utilise the available network bandwidth by comparing them to the OSU non-blocking put message rate microbenchmark. In this test, we configured the benchmark to use the same 10kB buffer size that conveyors were using internally, and then calculated how many updates per second it would translate to for histogram if we were able to achieve that level of throughput. This is effectively intended to give us an idea for a theoretical upper bound on aggregation performance potential and how wide the remaining gap may be at scale. We performed this test on two Frontier nodes in the same cabinet and connected via the same switch from 1–64 PPN in order to get an idea of the highest rates that could be achievable between two nodes, the results of which can be seen compared to that of contexts in Fig. 9.

At 64 PPN, we got a little under 200 million updates per second compared to the roughly 10–30 million we were seeing for conveyors and contexts. Considering the substantially greater overhead that aggregated processing incurs

compared to simply sending a lot of large messages, this is actually quite good performance. While the simulated message rate increases at lower PPN counts, this does not translate to similar increases for aggregation since it is bound by the cost of local processing of the much smaller packets, as was also previously made clear in Fig. 6. On the other hand, maximising the PPN count is typical, and the flat scaling behaviour we saw with aggregation suggests that the higher the PPN value, the closer we may be able to get to achieving the maximum performance potential of the NICs. This may be of benefit for systems with dense core configurations as are commonly used for data analytics workloads, though further investigation as such is left for future work.



**Fig. 9.** Simulated Message Rate via OSU.

### 5.3 Progress Model Optimisations

The final matter was discovered as an unintended consequence of a small optimisation we had made within aggregation contexts. Since in our case all aggregated messages are processed locally instead of remotely through remote direct memory access (RDMA) hardware, when PEs are instructed to perform operations on themselves, there is no need to pack and send messages through conveyors instead of processing it immediately and returning. However, we found that this effectively serialised conveyors’ progress management and resulting performance.

As described in Sect. 3.1, the progress model for conveyors is based upon filling of outgoing buffers rather than incoming ones. The problem is thus: if an application is never unable to push more data to other PEs, then it will never be forced to stop and process data that those other PEs are waiting on it for. It is effectively unable to voluntarily stop early either, since it has no insight from push as to the internal state of progress.

The reason we weren’t seeing this before when self-sending is because eventually with enough puts to oneself, a PE would fill its own outgoing buffers and be forced to stop pushing the next time it tries to self-send, triggering the pulling part of the progress loop. Without this, we can end up with all PEs endlessly pulling incoming data while waiting on the same source, which will then never be unable to send. This could conceivably create issues for applications employing

dynamic computation with heavy communication imbalances. Since this does not have a significant effect on the bottom line for our testing here and would require a bigger change within conveyors themselves, we are noting the issue here but a resolution for it is left for future work.

## 6 Conclusion

In this paper, we have demonstrated the effectiveness and scalability of conveyors and the aggregation context extension to OpenSHMEM on large-scale HPC systems. Through our experiments, we have shown that aggregation contexts can significantly improve the performance of independent AGP operations by reducing the overhead associated with small network message rates. Our results indicate that these aggregation strategies not only scale well with increasing node counts but also maintain consistent performance across both Slingshot 11 and InfiniBand interconnects.

Our analysis revealed that aggregation contexts, when built on top of conveyors, can achieve up to 65x performance improvements over traditional AGP operations on InfiniBand and up to 15x on Slingshot 11, where it can even outperform conveyors themselves by up to an additional 40%. The performance portability observed across different systems also suggests that this aggregation strategy is adaptable to different hardware configurations and interconnect technologies. This demonstrates the robustness and efficiency of this approach, making it a viable candidate for inclusion in the OpenSHMEM specification. Moreover, due to the high-level abstractions of aggregation, which allow applications to maintain their original structure, this strategy can be easily adapted to other interfaces with limited sets of communication operations, such as in MPI-3. Our findings also highlight areas for further optimisation, particularly in further reducing the overhead of aggregation and improving its progress management. Addressing these issues could lead to even greater performance gains and broader applicability of aggregation contexts in diverse HPC applications.

The aggregation context extension provides a scalable, performance-portable solution for enhancing small message rate performance in OpenSHMEM applications. By simplifying the incorporation of message aggregation with minimal burden on application code, this work holds promise for widespread adoption in the HPC community, enabling more efficient and effective use of modern high-speed interconnects. Future work will focus on refining these strategies and exploring their application to other emerging HPC systems and workloads.

**Acknowledgement.** This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally

sponsored research in accordance with the DOE Public Access Plan (<http://energy.gov/downloads/doe-public-access-plan>). This research used the Frontier and Andes resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. This work was funded through the Strategic Partnership Projects Funding Office via Los Alamos National Laboratory with IAN 61921590 for the project.

## References

1. De Sensi, D., Di Girolamo, S., McMahon, K.H., Roweth, D., Hoefer, T.: An in-depth analysis of the slingshot interconnect. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–14 (2020). <https://doi.org/10.1109/SC41405.2020.00039>
2. Jenkins, L., Zalewski, M., Ferguson, M.: Chapel aggregation library (CAL). In: 2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM), pp. 34–43 (2018). <https://doi.org/10.1109/PAW-ATM.2018.00009>
3. Khorassani, K.S., Chen, C.C., Ramesh, B., Shafi, A., Subramoni, H., Panda, D.K.: High performance MPI over the slingshot interconnect. *J. Comput. Sci. Technol.* **38**(1), 128–145 (2023). <https://doi.org/10.1007/s11390-023-2907-5>
4. Maley, F.M., DeVinney, J.G.: Conveyors for streaming many-to-many communication. In: 2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3), pp. 1–8 (2019). <https://doi.org/10.1109/IA349570.2019.00007>
5. Namashivayam, N., Cernohous, B., Pagel, M., Wichmann, N.: Early experience in supporting openSHMEM on HPE slingshot NIC (slingshot 11)
6. Paul, S.R., Hayashi, A., Chen, K., Sarkar, V.: A productive and scalable actor-based programming system for Å pgas applications. In: Groen, D., de Mulatier, C., Paszynski, M., Krzhizhanovskaya, V.V., Dongarra, J.J., Sloot, P. (eds.) *Computational Science - ICCS 2022*, pp. 233–247. Springer International Publishing, Cham (2022)
7. Poole, S.W., Curtis, A.R., Hernandez, O.R., Feind, K., Kuehn, J.A., Shipman, G.M.: *OpenSHMEM: Towards a Unified RMA Model*. Springer, New York, NY, USA, United States (2011). <https://www.osti.gov/biblio/1050391>
8. Venkatesan, V., Gorentla Venkata, M.: OpenSHMEM queues: an abstraction for enhancing message rate, bandwidth utilization, and reducing tail latency in openSHMEM applications. In: SC-W ’23, Proceedings of the SC ’23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, pp. 448–457. Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3624062.3624113>
9. Welch, A., Hernandez, O., Poole, S.: Extending openSHMEM with aggregation support for improved message rate performance. In: Cano, J., Dikaiakos, M.D., Papadopoulos, G.A., Pericàs, M., Sakellariou, R. (eds.) *Euro-Par 2023: Parallel Processing*, pp. 32–46. Springer Nature Switzerland, Cham (2023)



# Preliminary Study on Message Aggregation Optimizations for Energy Savings in PGAS Models

Oscar Hernandez<sup>1</sup>, Aaron Welch<sup>1</sup>(✉), Wendy Poole<sup>2</sup>, and Stephen Poole<sup>2</sup>

<sup>1</sup> Oak Ridge National Laboratory, Oak Ridge, TN, USA  
{oscar,welchda}@ornl.gov

<sup>2</sup> Los Alamos National Laboratory, Los Alamos, NM, USA  
{wkpoole,swpoole}@lanl.gov

**Abstract.** As Moore’s law has slowed down, we need to explore opportunities to save energy and power in parallel applications, especially during data movement. This is particularly true within the partitioned global address space (PGAS) model, where there is great potential for energy savings across distributed data accesses on large exascale computing systems. This paper explores the potential of message aggregation strategies within PGAS models, specifically focusing on OpenSHMEM, to improve energy efficiency on the CPU and memory of a node. Using the conveyor library, which aggregates small messages for network-efficient communication, we compare its performance gains in execution times against the energy reductions achieved. We compare applications from the bale effort as implemented through either atomic, get, and put or conveyor approaches on the Frontier supercomputer. Our preliminary results show significant improvements in both performance and energy consumption. These findings suggest that message aggregation can play an important role in addressing the challenges of PGAS energy consumption in modern HPC systems.

**Keywords:** PGAS programming models · OpenSHMEM · message aggregation · energy efficiency

## 1 Introduction

Due to the diminishing returns of Moore’s Law and Dennard scaling, HPC leadership computing systems are hitting a power wall that must be addressed through new architectures and the co-design of multiple layers in system software and hardware. Traditional modeling and simulations still require significant compute resources and power to scale, to implement new first-principles models and/or increase their resolution for scientific discoveries. This need is exacerbated by emerging AI workloads and the new scaling properties of neural networks, which optimize loss functions to train safe and trustworthy AI models. As a result,

new architectures are starting to emerge that are more specialized and energy-efficient, with novel memory interconnects aimed at providing access to local, neighboring, and global memories required for model parallelism and to distribute data loading stages.

PGAS models, which traditionally have focused on addressing HPC data-centric computing requirements for applications, have the potential to optimize various aspects of AI pipelines by efficiently distributing data in local and remote memories and accessing them with atomic, get, and put (AGP) operations, mitigating the energy cost of data movement [4]. The PGAS model can also benefit from scheduling work where the data is located by exploiting data locality to improve application performance. However, with remotely accessible memory, bursts of small data accesses can lead to congestion due to message rate limitations in the network interconnect, resulting in inefficient use of computing resources as processes wait for communication to complete. This can lead to energy waste due to longer wait times, excessive communication progress checks, and the use of resources that could otherwise be utilized for computation.

An area that has been explored to address this small message rate problem is message aggregation. Message aggregation involves grouping operations and their values and sending them in bulk to their destinations for completion on the remote end, similar to the active message communication model. This approach leverages faster local memories on both local and remote nodes to process and aggregate messages, using the network bandwidth more efficiently to overcome the message rate issues in today’s interconnect technologies. Aggregation has shown significant speedups in many-to-many communication patterns relevant to data analytics, graph processing, irregular data accesses, and sparse data computation patterns. The conveyor library’s aggregation approach [5] has shown promising results that can be leveraged by high-level PGAS programming models such as Chapel [3], UPC [1], and OpenSHMEM [7].

Our paper aims to address the following key questions regarding PGAS and small message aggregation strategies:

- How much energy can be saved when using a message aggregation strategy compared to the traditional AGP model of a PGAS application?
- How much of the message aggregation performance improvements translate into energy savings, and do they scale at the same rate as performance?
- How do message aggregation optimizations translate to power utilization on the CPU and memory?

Our work-in-progress aims to answer these questions by understanding how the performance gains provided by a message aggregation strategy translate to energy and power consumption, and how these gains can help free computing resources to address current power and energy challenges in HPC applications.

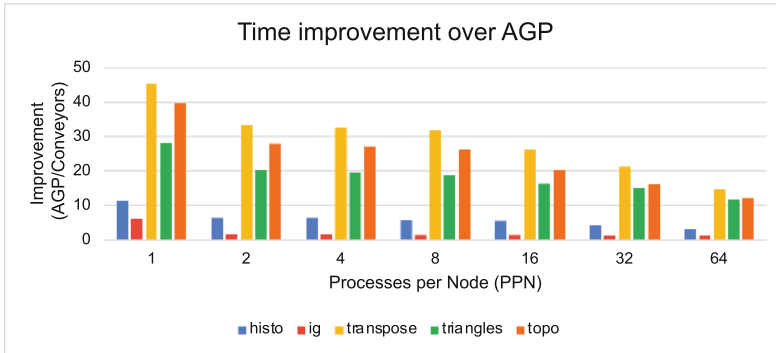
## 2 Preliminary Studies

We used the Oak Ridge National Laboratory’s current supercomputing flagship, Frontier, to start answering our questions. Frontier has a theoretical peak



double-precision performance of approximately 1.7 exaflops and an overall power consumption of approximately 22.7 megawatts. The system has 74 HPE Cray EX Olympus racks, each housing 128 AMD compute nodes, totaling 9,472 compute nodes. Each Frontier compute node includes a 64-core AMD Optimized 3rd Gen EPYC CPU with 512 GB of DDR4 memory. Additionally, each node contains four AMD MI250X acspGPU, each with two Graphics Compute Dies (GCDs) and 64 GB of high-bandwidth memory (HBM2E). Frontier has a state-of-the-art Slingshot 11 interconnect with network congestion management, dynamic routing, and quality of service (QoS) protocols in their Rosetta switches [2]. Our study focused on the energy and power consumption of the CPU and memory for PGAS communication operations using the Cray implementation of OpenSHMEM [6]. We leveraged the counter information provided by HPE at the node level, capturing accumulated energy (in joules) and point-in-time power (in watts) for the entire node, the CPU socket, and memory, accessible via `/sys/cray/pm_counters`.

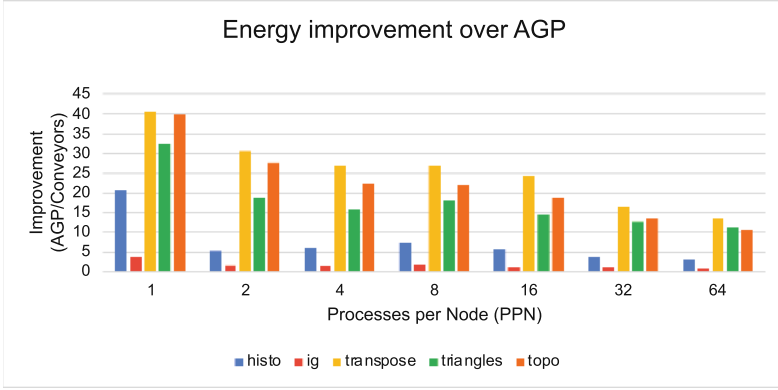
Our preliminary evaluation focused on the bale 3.0 applications, which implement several many-to-many communication patterns written using both the AGP and conveyor aggregation models. We ran these applications using their default input parameters on two nodes of Frontier.



**Fig. 1.** Time improvements of message aggregation over the AGP model.

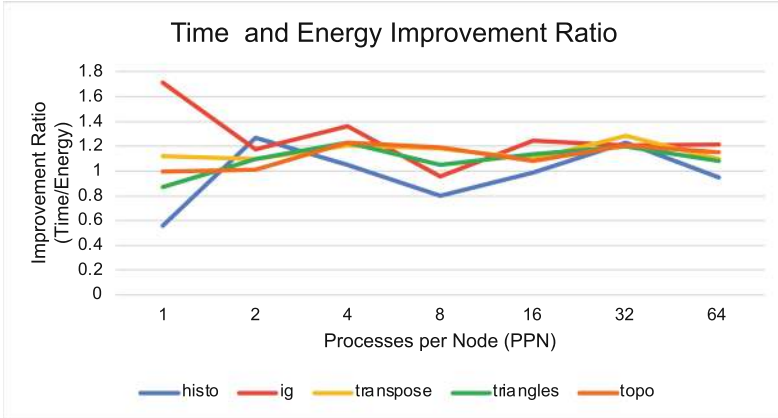
Figure 1 shows the time improvements of the bale applications using conveyors over the AGP model from 1–64 processes per node (PPN). All the bale applications benefit from conveyors, especially sparse matrix transpose, triangle counting, and topological sort, which can achieve improvements of 45x, 28x, and 40x for 1 PPN, respectively. As we weak scale the program sizes and increase the number of processing elements (PEs), there are still significant aggregation performance gains of 15x, 12x and 12x when using 64 PPN for these applications.

Figure 2 shows the combined CPU and memory energy improvements when using conveyors over the AGP versions. Here, we see energy consumption improvements of 41x for sparse matrix transpose, 32x for triangle counting, and



**Fig. 2.** Energy improvement of aggregation over the AGP model.

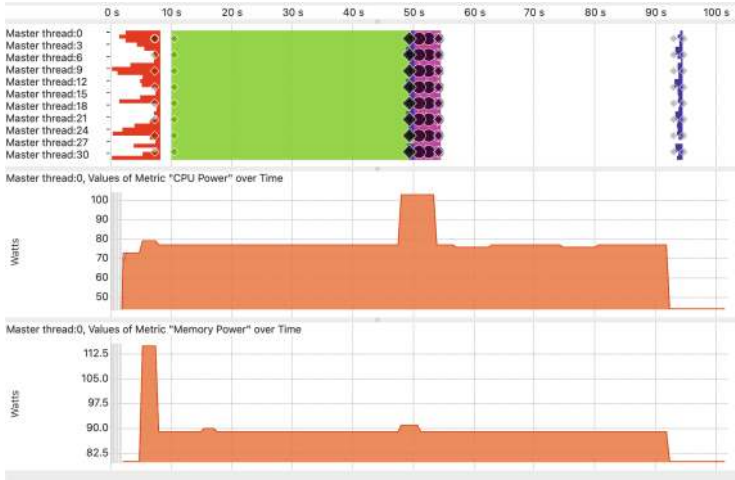
39x for topological sort for 1 PPN. As we weakly scale the problem sizes, we again see significant energy improvements for these applications, with 13x, 11x, and 10x improvements for 64 PPN, respectively.



**Fig. 3.** Ratio of time and energy improvement of message aggregation over AGP.

Figure 3 shows the ratio of performance and energy improvement as we scale the bale applications up to 64 PPN. Anything above 1 indicates that the application improves faster in performance than energy; anything below 1 means the application improves better in energy compared to performance. We noticed that several applications, such as histogram and triangle counting, have better energy improvements over time at 1 PPN. At different PPN values, we see different ratios of improvements. At lower PPN counts, the majority of the energy consumption for conveyors comes from the memory—around 58%. As we increase

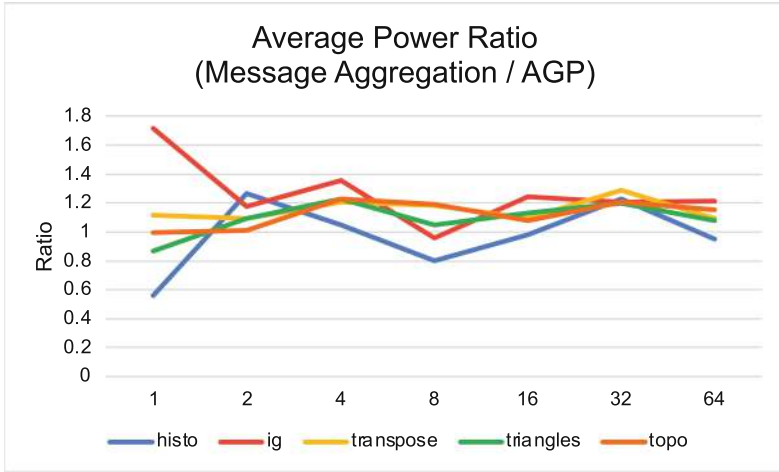
the PPN count, the cores in the socket start to consume more energy—using 66% of the total energy at 64 PPN for the histogram application. For triangle counting this is 60% and 66% respectively. A similar trend is seen in both the AGP and conveyor versions. It is noteworthy that conveyors use a higher percentage of energy from memory compared to the AGP version, due to performing more memory operations for aggregation. The other observation is that the energy source is more balanced at 8–16 PPN, where the memory and CPU socket contribute a similar amount (around 50%) of the total energy. An interesting observation is that for histogram and triangle counting, energy consumption increases at a faster rate than performance at 1 PPN, with most of the energy being consumed by the memory.



**Fig. 4.** Power consumption for AGP and message aggregation histogram.

Figure 4 shows a trace of the power consumption for the AGP and conveyor versions of histogram on a single run when running at 16 PPN. The red lines represent the time spent on initialization using `shmem_init`. The green lines represent the AGP version, while the purple areas indicate the conveyor version. The purple lines represent time spent in `shmem_barrier_all`. The empty white region at the end is the verification step of the application. From the trace, we notice that the power consumption for AGP remains constant throughout the execution at 77 W. When it starts executing the conveyor version, the power consumption increases to a constant 103 W for the CPU. The power consumption for the memory also remains relatively constant at 89 W for AGP and 91 W for conveyors. As the conveyor version executes much faster, its overall energy consumption decreases despite the increase in power.

Figure 5 shows the average power ratio of the conveyor versions over AGP for the bale applications. Anything greater than one indicates that the average power



**Fig. 5.** Combined CPU and memory average power ratio of message aggregation over AGP.

consumption of the conveyor version is higher than that of AGP. The majority of the conveyor versions consume more power than their AGP counterparts, with exceptions for histogram and triangle counting at 1 PPN, histogram and topological sort at 8 PPN, and histogram at 16 PPN. The trend is that the conveyor versions use more CPU power compared to the AGP versions as we increase the number of cores used per node.

### 3 Conclusions

This study provides a preliminary analysis of the impact of message aggregation strategies on power and energy consumption for PGAS programming models. The use of the conveyor library for message aggregation demonstrates significant performance improvements that are also coupled with significant energy savings, achieving savings of up to 41 times. In certain cases, such as histogram and triangle counting, energy savings improvements surpass performance improvements at 1 PPN.

Aggregation strategies also alter the balance of energy consumption between the CPU and memory. Initially, memory operations consume more energy for lower PPN counts, but overall energy utilization becomes more balanced around 8–16 PPN. Beyond this point, energy consumption is dominated by the CPU cores. In general, the preliminary results indicate that message aggregation can significantly improve both performance and energy consumption at the cost of increasing power for both the CPU and memory. This makes it a valuable approach to optimize PGAS models for energy consumption by better utilizing hardware resources in high-performance computing systems.

**Acknowledgments.** This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (<http://energy.gov/downloads/doe-public-access-plan>). This research used the Frontier and Andes resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. This work was funded through Strategic Partnership Projects Funding Office via Los Alamos National Laboratory with IAN 619215901 on the project “OpenSHMEM - Standardized API for parallel programming in the Partitioned Global Address Space”.

## References

1. Alvanos, M., Farreras, M., Tiotto, E., Amaral, J.N., Martorell, X.: Improving communication in PGAS environments: static and dynamic coalescing in UPC. In: ICS '13, Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, pp. 129–138. Association for Computing Machinery, New York, NY, USA (2013). <https://doi.org/10.1145/2464996.2465006>
2. De Sensi, D., Di Girolamo, S., McMahon, K.H., Roweth, D., Hoefer, T.: An in-depth analysis of the slingshot interconnect. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–14 (2020). <https://doi.org/10.1109/SC41405.2020.00039>
3. Jenkins, L., Zalewski, M., Ferguson, M.: Chapel aggregation library (CAL). In: 2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM), pp. 34–43 (2018). <https://doi.org/10.1109/PAW-ATM.2018.00009>
4. Kestor, G., Gioiosa, R., Kerbyson, D.J., Hoisie, A.: Quantifying the energy cost of data movement in scientific applications. In: 2013 IEEE International Symposium on Workload Characterization (IISWC), pp. 56–65 (2013). <https://doi.org/10.1109/IISWC.2013.6704670>
5. Maley, F.M., DeVinney, J.G.: Conveyors for streaming many-to-many communication. In: 2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3), pp. 1–8 (2019). <https://doi.org/10.1109/IA349570.2019.00007>
6. Namashivayam, N., Cernohous, B., Pagel, M., Wichmann, N.: Early experience in supporting openSHMEM on HPE slingshot NIC (slingshot 11)
7. Welch, A., Hernandez, O., Poole, S.: Extending openSHMEM with aggregation support for improved message rate performance. In: Cano, J., Dikaiakos, M.D., Papadopoulos, G.A., Pericàs, M., Sakellariou, R. (eds.) Euro-Par 2023: Parallel Processing, pp. 32–46. Springer Nature Switzerland, Cham (2023)

# Author Index

## B

Budimlić, Zoran 50

## C

Cogumbreiro, Tiago 55

## D

Ding, Shuo 1

## E

Elmougy, Youssef 11, 70

## G

Gressmann, Frithjof 78

## H

Harrison, Robert J. 50

Hayashi, Akihiro 11, 64, 70

Hernandez, Oscar 103, 114

## J

Javanmard, Mohammad Mahdi 50

## K

Kasahara, Hironori 34

Kawasumi, Tohma 34

Kimura, Keiji 34

Knoke, Kathleen 50

## L

Lange, Julien 55

## M

Mikami, Hiroki 34

Mysore, Aniruddha 11

## P

Pickar, John 34

Pollard, Samuel D. 22

Poole, Stephen 103, 114

Poole, Wendy 103, 114

Pouchet, Louis-Noël 50, 90

## R

Rauchwerger, Lawrence 78

## S

Shirako, Jun 64

Singhal, Shubhendra Pal 70

Sullivan, Zachary J. 22

## T

Tucker, Emily 90

## W

Welch, Aaron 103, 114

## Y

Yang, Jiawei 70

## Z

Zhang, Qirun 1

Zhou, Tong 64