MPI Errors Detection using GNN Embedding and Vector Embedding over LLVM IR

Jad El Karchi[2], Hanze Chen[1], Ali TehraniJamsaz[1], Ali Jannesari[1], Mihail Popov[2], Emmanuelle Saillard[2]

[1]*Iowa State University, Ames, Iowa, USA*
[2]*Inria, Bordeaux, France*
{hanzech, tehrani, jannesar}@iastate.edu
{jad.el-karchi, mihail.popov, emmanuelle.saillard}@inria.fr

Abstract—Identifying errors in parallel MPI programs is a challenging task. Despite the growing number of verification tools, debugging parallel programs remains a significant challenge. This paper is the first to utilize embedding and deep learning graph neural networks (GNNs) to tackle the issue of identifying bugs in MPI programs. Specifically, we have designed and developed two models that can determine, from a code's LLVM Intermediate Representation (IR), whether the code is correct or contains a known MPI error.

We tested our models using two dedicated MPI benchmark suites for verification: MBI and MPI-CorrBench. By training and validating our models on the same benchmark suite, we achieved a prediction accuracy of 92% in detecting error types. Additionally, we trained and evaluated our models on distinct benchmark suites (e.g., transitioning from MBI to MPI-CorrBench) and achieved a promising accuracy of over 80%. Finally, we investigated the interaction between different MPI errors and quantified our models generalization capabilities over new unseen errors. This involved removing errors types during training and assessing whether our models could still predict them. The detection accuracy of removed errors vary significantly between 20% to 80%, indicating connected error patterns.

Index Terms—Deep learning, Verification, MPI, GNN

I. INTRODUCTION

High-Performance Computing (HPC) plays an important role in many fields like health, materials science, security, and the environment. The current supercomputer hardware trends lead to more complex parallel applications with heterogeneity in hardware, new scalable algorithms, and combinations of parallel programming models that pose many programmability challenges. This demands a requirement for more efficient and scalable debugging techniques to assist HPC application developers and parallel programming. Yet, despite the growing number of verification and debugging tools, determining if a parallel program always behaves as expected on any execution is challenging due to non-deterministic executions [1].

MPI is one of the most popular programming models in High-Performance Computing. In an MPI program, each MPI process executes a parallel instance of a program in a private address space and exchanges data across distributed memory systems via messages. MPI exposes many ways of exchanging data, including collectives, point-to-point, persistent, and one-sided communications: many errors can occur in an MPI program.

This paper proposes a new AI-assisted approach for detecting errors in MPI programs. In particular, we devise a machine learning approach that leverages the representational learning of programs to identify errors. The experimental results indicate that the approach is highly effective in identifying different types of errors or assessing if a code is correct. In summary, this paper makes the following contributions:

- ML approaches with MPI errors detection capabilities that are on par with existing verification tools. They only require a labeled code dataset and can be applied to new scenarios.
- A thorough evaluation of the error detection mechanisms over two dedicated benchmark suites: the MPI Bugs Initiative (MBI) [2] and MPI-CorrBench [3].
- A quantification of the generalization of our models by predicting across benchmark suites, by removing errors in the training set and looking for them in the validation, and by studying an error in a real application.

The rest of the paper is structured as follows: The next section discusses related works. Section III describes the datasets we use for our experiments, while Section IV explains the details of the proposed approach. Section V provides experimental results. Section VI discusses our limitations and future work.

II. RELATED WORKS

A. MPI Verification Method

Related works on MPI program verification use many approaches to detect errors in MPI applications, including static analysis, symbolic execution, concolic testing, model checking, dynamic verification techniques, MPI special libraries and trace-based approaches. Because of the large diversity of programming features and complexity of current systems, no existing tool is currently able to detect all possible errors [2], [3]. They all come with restriction: they are focused on one programming model, they target a specific error, they work with a specific MPI implementation, or they are not yet mature.

Static analyses enable an early errors detection (i.e., the program is not executed) but can report false positives (an error is reported on a correct scenario). Among static tools, MPI-Checker [4] is based on the Clang Static Analyzer. It performs so-called AST-based and path-sensitive checks. AST-based checks include correct type usage while path-sensitive checks verify aspects of nonblocking communication, based on the usage of MPI requests. CIVL [5] and MPI-SV [6] both

combine symbolic execution and model checking to detect communication deadlocks. MPISE and Hermes [7] detect communication deadlocks with concolic testing. This method, proven efficient on sequential programs, performs symbolic execution dynamically with a concrete execution [8], [9]. Compared to these two tools, COMPI [10], [11] uses input tuning to achieve high branch coverage and tackles runtime bugs like assertion violation or infinite loops. SimGridMC [12] and ISP [13] check if a program satisfies a given property (e.g., liveness, communication determinism) by considering all possible executions. Like all model checkers, they face the state space explosion problem. Aislin [14] is an explicit-state model checker which verifies MPI programs with arbitrary-sized system buffers. Dynamic verification techniques, such as MUST [15] detect runtime errors. These tools find real errors but stand for a specific environment and can miss errors (false negatives). MUST intercepts all MPI operations to perform online checking. It is based on GTI [16] (Generic Tool Infrastructure) and can detect multiple errors like deadlocks, type mismatches or resource leaking. DAMPI [17] and Intel Trace Analyzer and Collector (ITAC) [18] detect deadlocks with a time-out approach. MC-CChecker [19] and MC-Checker [20] both use a trace-based approach to detect memory consistency. They focus on MPI Remote Memory Access (RMA) correctness and do not support other MPI features. Validation can also be done inside MPI libraries such as in MPICH [21] or NEC-MPI [22]. However, the detection of errors is limited to the information available to the MPI routines. PARCOACH [23], [24] combines static analysis with code instrumentation to detect misuse of MPI collectives and data races that can occur when using nonblocking and persistent communications as well as one-sided communications [25]–[27]. Although it uses a precise data- and control-flow interprocedural analysis to pinpoint root cause problems, it may lead to many false positives.

All tools cited are using their own terms to report errors and are subject to runtime and compilation failure if a feature is not supported by the tool. In this paper, we propose a new method that learns incorrect patterns by studying the source code (i.e., the compiler Intermediate Representation specifically), irrespective of the language or the MPI implementation used. Our method detects all errors present in the benchmark dataset. We also investigate the importance of each MPI error type with an ablation study that removes errors from the learning dataset and tries to detect them during validation.

B. ML for Bug Detection

Novel ML techniques have emerged for bug detection and code refactoring. They efficiently detect and potentially correct issues without humans costly hand-crafting detectors (e.g., variable misuse [28], wrong binary operator [29], comment deletion in Python [30], or specific Javascript functions [31]). The insight is to associate patterns in the source code to identified bugs with Deep Neural Network (DNN)-based models. To check a new code, the DNN just needs to search for these patterns within the source code. While the source code is directly written by the developer, it is not the only used code representation to detect patterns. Intermediate structures manipulated by the compiler such as the Abstract Syntax Tree (AST), a tree representation of the abstract syntactic structure of a source code, or the LLVM Intermediate Representation [32] (IR), an accurate and language/hardware independent code representation potentially used with a virtual machine, are also used by ML techniques [33]–[37]. Nevertheless and to the best of our knowledge, despite this diversity of representations, ML techniques currently fail to detect bugs in parallel programs either because the representations fail to efficiently expose the contexts of the patterns or because of insufficient training data.

C. Vulnerability Detection

In software developments, vulnerabilities are those security flaws or weaknesses that attackers can exploit. There have been works that use expert-defined insights [38] or symbolic execution [39], [40] to identify vulnerabilities. Recently, machine learning has also been used for detecting vulnerabilities [41]–[43]. While vulnerability detection has benefited from machine learning-based approaches, MPI error detection, on the other hand, has not received the same attention. One of the reasons is that MPI errors are mostly non-deterministic and hard to identify. In this paper, we aim to fill in this gap by targeting MPI errors and leveraging the latest techniques in machine learning.

To the best of our knowledge, we propose the first method using ML to detect errors specifically in MPI applications.

III. DATASETS

We created datasets to train ML models by using two MPI benchmarks suites that have recently emerged: the MPI Bugs Initiative (MBI) [2] and MPI-CorrBench [3]. These benchmarks are the only MPI correctness benchmarks and are integrated into state-of-the-art MPI verification tools such as MUST and PARCOACH.

MBI contains almost 2,000 codes written in C. The initiative proposes 9 types of errors, gathered according to which context they manifest: *single call*: invalid parameter, *single process*: resource leak, request lifecycle, epoch lifecycle and local concurrency, and *multi-processes*: parameter matching, message race, call ordering, and global concurrency. Each code in MBI has a header describing the error in the code, how to execute it and what MPI features are used. MPI-CorrBench contains around 400 small codes (referred as level zero) written in C. MPI-CorrBench follows a different error classification compared to MBI. MPI-CorrBench errors can be either erroneous arguments (ArgError), mismatching arguments (ArgMismatch) or erroneous program flow (MissplacedCall and MissingCall). Unlike MBI, codes in MPI-CorrBench do not have a header specifying the errors. Therefore, we relied upon programs names to associate a program with its error type (e.g., the code `ArgError-MPIIRecv-Count-1.c` is associated with the ArgError error). Both benchmark suites also include a substantial number of correct codes, which is
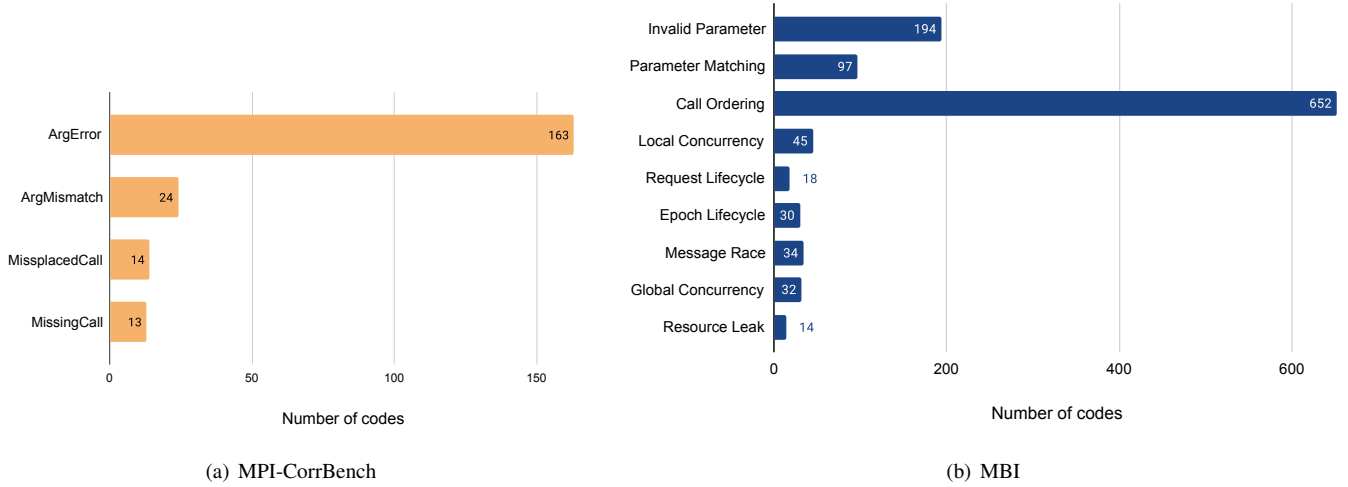
(a) MPI-CorrBench

(b) MBI

Fig. 1. Number of codes per error type in MPI-CorrBench (left) and MBI (right).

crucial to test our models across a range of different contexts and use cases.

Figure 3 shows the number of correct and incorrect codes in each benchmark. The distribution among incorrect codes is presented in figure 1. In both benchmarks, one type of error is more represented: call ordering for MBI and ArgError for MPI-CorrBench. To gain a deeper understanding of the structure and composition of the two benchmarks, we use a violin plot, presented in figure 2, to visualize the number of lines in each individual code in the benchmarks. In MBI, we observe that the codes are relatively similar in length. However, in MPI-CorrBench, the violin plot shows a wider range of chord lengths, with some codes consisting of just a few lines and others containing much larger and more complex structures. In particular, unlike incorrect codes, correct codes have at least 10^3 lines of code in MPI-CorrBench. This raises concerns that the model might be biased toward longer codes to identify correct codes. Code length is a poor criteria to predict the presence or absence of errors (or at least not an absolute one). To address this issue, we applied several methods to remove the bias so that the model can accurately distinguish between correct and incorrect codes. Specifically, correct codes include an extra header *"mpitest.h"* in MPI-CorrBench, which is not necessary to compile the applications and adds all the extra lines during the pre-processing: we simply removed this call across the different codes. We also consider various compilation options and normalization methods that we further describe in Section V-A.

In this paper, we consider three different datasets for our models: MBI, MPI-CorrBench without bias due to code size (to which we refer simply as MPI-CorrBench for the rest of the paper), and the combination of MBI with MPI-CorrBench, referred as *Mix* in the rest of the paper.

IV. MACHINE LEARNING METHODS

This section presents how we train and apply our models over the datasets presented in the previous section. To predict if an application is correct, our model needs to be trained over a set of codes that are represented each with a set of features along with an error label. The datasets already label each code with an error type or identify it as correct. Different levels of programs representations such as Abstract Syntax (AST), control flow, or data flow are valid options to extract the features. We select the LLVM Intermediate Representation (IR) as input for our models since it proved to be a successful representation to expose information for Deep Learning (DL) models when optimizing various tasks [37], [44]. The rest of this section focuses on how each of our model uses the IR along with the dataset labels to train models.

A. *IR2vec embedding*

Our first strategy consists in exposing the IR to ML models through embedding with IR2vec [37]. Figure 4 presents the detailed workflow of how we train and validate the IR2vec based prediction model. IR2vec transforms IR code into a vector embedding in a continuous space. The underlying assumption behind this transformation is that *similar* applications should result in vectors that are close to each other. IR2vec was previously applied for optimizing heterogeneous task mapping and thread coarsening but not for verification: In this work, we extend the IR embedding for error detection.

IR2vec proposes 2 encoding strategies, symbolic and flow aware. The former performs a seed embedding while the latter performs a seed embedding but also augments it with flow-aware information. Each encoding generates a vector of 256 elements per IR compilation unit.

Because the cost of inferring the embedding is negligible compared to executing an MPI application, we executed both the symbolic and flow aware encodings and concatenated them into a single vector. We use this vector as feature input for a
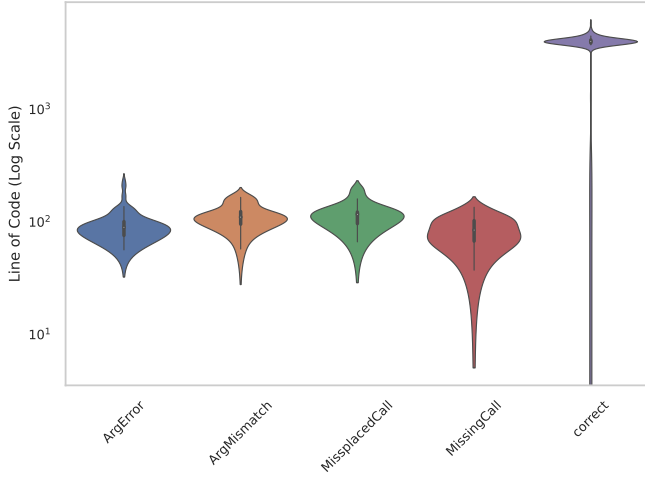
(a) MPI-CorrBench
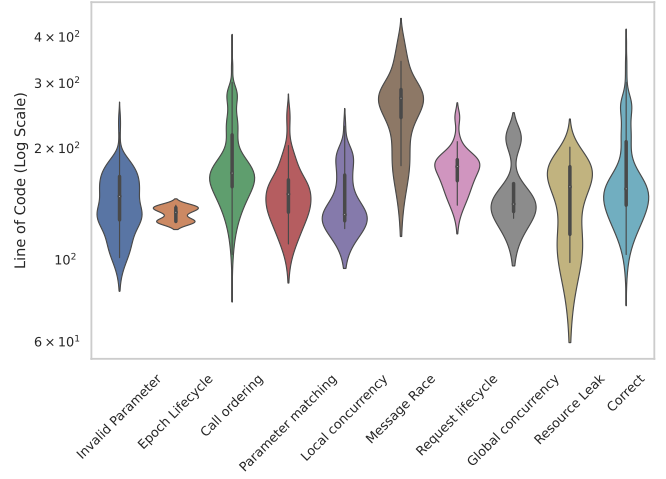
(b) MBI

Fig. 2. Code size in MPI-CorrBench (left) and MBI (right). The line of code is reported after performing the C pre-processing include calls. MPI-CorrBench correct codes have a high line count compared to the incorrect codes. On the opposite, MBI has no significant outlier in the line count.
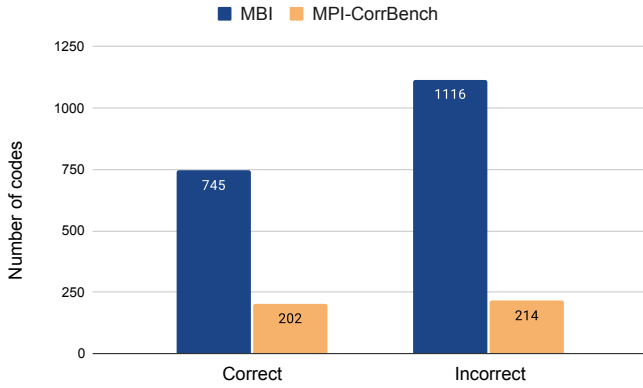

Fig. 3. Number of correct and incorrect codes in MBI and MPI-CorrBench.

ML classifier to determine if a code is correct or contains a specific error. In particular, we provide the concatenated embedding vector to a Decision Tree (DT). The labels of the decision tree are simply the different types of error described in Section III (or a description of whether the code is correct or not) while the features are the vector embeddings for each code. The DT uses the default Scikit learn setup [45] (version 1.0).

To remove the noise in the feature vectors, we perform a feature selection step with Genetic Algorithms (GAs). Each individual is considered as a subset of vector coordinates. The fitness function of the individual is the quality of the subsequent prediction model. We mutate individuals by changing which vector coordinates are selected. In total, we consider a population size of 2500 individuals, with 25 generations, 90% and 10% crossover and mutation probabilities, respectively, where each individual is composed of 5 vector coordinates.

The GA was implemented with *pyeasyga* [46] (version 0.3.1). Section V-C evaluates the benefits of this step.

B. Graph Neural Network Embedding

Our second strategy utilizes graphs and Graph Neural Networks (GNNs) to identify MPI bugs as shown in Figure 5. A lot of software analyses are performed over graphs, such as control flow and data flow. Therefore, representing programs as graphs allows us to present such flow-aware information to the neural network models. To classify programs using GNNs, we adapt ProGraML [44] representation, which is a program graph representation built on top of LLVM IR. It specifically creates data flow, control flow, and call graphs and presents them in one unified graph. As a result, we have three types of edges or relations between nodes. To effectively model different relations and nodes, we treat each graph as a heterogeneous graph with three types of nodes (i.e., control, variable, constants) and three types of edges (i.e., control, data, call) and use HeteroConv layers in PyTorch Geometric [47] to support heterogeneous graphs. We use Graph Attention Convolution (GATv2) [48] as our graph convolution layer.

Therefore, our GNN-based MPI error detection pipeline is as follows: We first generate ProGraML representation for each sample in our dataset, then feed these graphs to 3 consecutive GATv2 layers with the sizes of 128, 64, and 32 respectively. After applying GATv2 layers, we have a latent representation (vector) of size 32 for each node in the graphs. Then we apply an adaptive max pooling layer to aggregate the latent representation of all nodes into one vector. As a result, we would have one vector per each graph (graph-level vector). Then, graph-level vectors are passed to two fully connected layers. The output dimension of the last fully connected layer corresponds to the number of classes that we have in our training set.
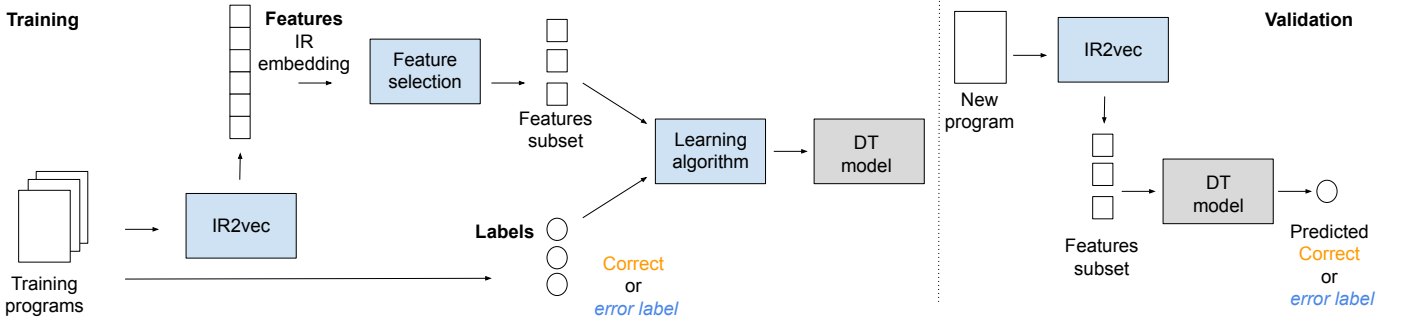
Fig. 4. Predicting errors in MPI applications with embedding based models.

We use the cross-entropy loss function to measure the error rate of the GNN model during training and use the Adam optimizer with the learning rate of 4×10^{-4} to update the weights of the model to minimize the loss value. The GNN pipeline is trained for 10 epochs.

V. EXPERIMENTAL RESULTS

This section presents the results of our two models on the datasets, a comparison with related works, and an ablation study for generalization purposes. We used MBI (commit *1ab5a546*) and MPI-CorrBench version 1.2.1. Our results are reproducible at https://gitlab.inria.fr/reproducibility/paper-mpi-errors-detection-using-gnn-embedding-and-vector-embedding-over-llvm-ir-reproducibility.

We designed different evaluation scenarios to incrementally increase the difficulty of detecting errors: *Intra*, *Mix* (previously described in Section III), and *Cross*. **Intra** (see Section V-A) consists in training and evaluating models on either MBI or MPI-CorrBench while **Mix** (see Section V-B) consider both benchmark suites together. Such scenarios are helpful to determine if ML based approaches are actually capable of detecting errors even on applications that come from the same benchmark suite and thus share some code structures. To avoid over-fitting, we used a standard *10-fold cross-validation* to evaluate our models over Intra and Mix. Thus, for each dataset, ML method (e.g., IR2vec and GNN), and Intra and Mix, we train ten models and evaluate each one over validation folds composed of approximately 200, 40, and 240 codes (i.e., 10%) for MBI, MPI-CorrBench, and mix, respectively. For the rest of the paper, all the prediction results over Intra or Mix are an aggregation of the 10 validation folds.

Finally, **Cross** (see Section V-C) refers to train a model on a benchmark suite and evaluate it on a distinct benchmark suite (e.g., train model on MBI and validate it on MPI-CorrBench). This scenario demonstrates the generalization capability of our models by evaluating them on significantly different code structures and errors: not only are the code structures changing between benchmark suites, but so are the labelled errors (as discussed in Section III). This is particularly challenging for the models as the different datasets might literally not contain some errors, limiting the potential accuracy of our approach. To train and evaluate the same model over different datasets,

we updated the labeling of each code to either correct or incorrect.

To assess the quality of our predictions when considering correct or incorrect labels, we used a set of metrics that Table I summarizes. Each metric is computed with the number of true positive *TP* (error correctly detected), true negative *TN* (correct code reported as such), false positive *FP* (correct code reported as faulty) and false negative *FN* (error missed). The first part of the table gives the most used metrics in the state-of-the-art while the second part of the table gives metrics defined in MBI.

A. Intra Modeling

We first trained and validated our models on a standalone benchmark suite (MBI or MPI-CorrBench). End results of correct and incorrect code predictions are presented in Table II under the lines *IR2vec Intra* and *GNN Intra*. To achieve these results, we explored different parameters in the models. In particuar, we consider compiler options for both approaches, and normalization, prediction labels, feature selection, and seeds for IR2vec. These are parameters in our methods that can be fine-tuned to increase the overall accuracy or provide more insights.

Compilation options. Both GNN and IR2vec methods use IR as input. It is interesting to note that the compiler can optimize the IR with different compiler passes (e.g., *-O2*, *-O3*) to improve performance. However, recent studies [49], [50] demonstrated that compiler passes also enable to expose more information to DNNs. DNNs [49], [50] take as input IR and use it for parameter optimization such as device mapping or NUMA setting. In other words, using custom compiler passes with DNNs improve the prediction capabilities over a single compiler sequence when generating the IR. In our work, we considered different compiler options and selected two in particular for the rest of this work: *-O0* for GNN and *-OS* for IR2vec. Our intuition is that since *-O0* leaves the code intact, it may ease the error detection. However, some errors may only occur when the code is optimized (i.e., a bug might not be visible, but optimizations may trigger its manifestation): there is a trade-off between studying a simplified code versus one complex that is representative of the real execution. We selected *-OS* for IR2vec to reduce the impact
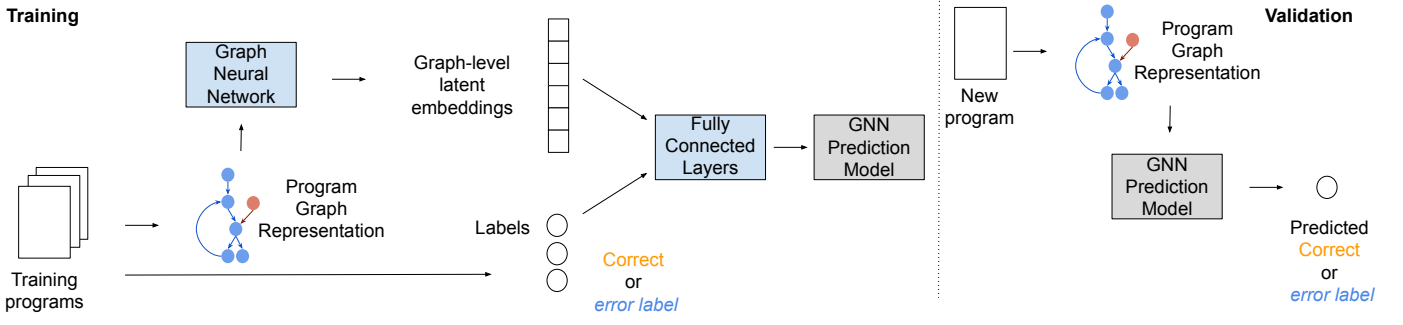
Fig. 5. Graph Neural Network based model to predict errors in MPI.

Metric	Definition	Meaning
Recall	$R = \frac{TP}{TP+FN}$	Ability to find existing errors
Precision	$P = \frac{TP}{TP+FP}$	Potential confidence when a code is reported as correct
F1 Score	$F1 = \frac{2 \times P \times R}{P+R}$	Overall bug-finding quality
Accuracy	$A = \frac{TP+TN}{Total}$	Proportion of correct diagnostics over the tests
Coverage	$Cov = 1 - \frac{CE}{Total+Errors}$	Ability to compile codes
Conclusiveness	$Cc = 1 - \frac{Errors}{Total+Errors}$	Ability to draw a diagnostic on codes
Specificity	$S = 1 - \frac{TN}{TN+FP}$	Ability to not find errors in correct codes
Overall accuracy	$Oa = \frac{TP+TN}{Total+Errors}$	Proportion of correct diagnostics over all tests

TABLE I

METRICS USED IN THE EVALUATION. CE = COMPILATION ERROR, TO = TIME OUT, RE = RUNTIME ERROR, TP = TRUE POSITIVE, TN = TRUE NEGATIVE, FP = FALSE POSITIVE, FN = FALSE NEGATIVE. TOTAL = TP+FP+TN+FN , ERRORS = CE+RE+TO. THE SECOND PART OF THE TABLE DEPICTS METRICS DEFINED IN [2].

Model	Dataset		Results				Metrics			
	Training	Validation	TP	TN	FP	FN	Recall	Precision	F1 Score	Accuracy
IR2vec Intra	MBI	MBI	1043	664	81	73	0.935	0.928	0.931	0.917
	CORR	CORR	200	184	18	14	0.934	0.917	0.925	0.923
IR2vec Cross	MBI	CORR	182	176	32	26	0.875	0.850	0.862	0.860
	CORR	MBI	805	523	311	222	0.784	0.721	0.751	0.713
IR2vec Mix	MBI + CORR	MBI + CORR	1198	811	136	132	0.901	0.898	0.899	0.882
GNN Intra	MBI	MBI	1045	657	88	71	0.922	0.936	0.929	0.914
	CORR	CORR	180	154	22	60	0.719	0.875	0.79	0.803
GNN Cross	MBI	CORR	189	168	34	25	0.832	0.870	0.851	0.858
	CORR	MBI	1108	19	726	8	0.604	0.993	0.751	0.605
GNN Mix	MBI + CORR	MBI + CORR	1228	846	101	102	0.893	0.892	0.893	0.911

TABLE II

RESULTS OF OUR MODELS ON THE THREE DATASETS. CORR = MPI-CORRBENCH. ALL THE PREDICTIONS ARE ON WHETHER THE CODE IS CORRECT OR INCORRECT.

of different code sizes: our assumption is that *-OS* will reduce the IR size difference between different codes. We empirically evaluated *-O0* (easy to analyze), *-O2* (representative), and *-OS* (reduced bias due to size) in Table IV. At most, compiler optimizations improve the accuracy by approximately 5% over MPI-CorrBench.

Normalization. IR2vec generates vectors as input to a DT that actually predicts the code correctness. Yet, naively using this vector can bias the prediction. For instance, we observe that long codes tend to generate larger vectors. This was particularly a problem when studying MPI-CorrBench correct codes: vectors with large values were systematically referring to correct codes due to the bias in the dataset before removing

the headers. To further remove such artifacts, we considered two normalization strategies. We either normalized each vector to contain values between 0 and 1 by dividing each element with the max of the vector or we normalized each vector coordinate across all the codes. We ended up using the former normalization as it ensures that every code has a vector with values between 0 and 1 independently of its size or any other code. Table IV presents in details the different normalization strategies over MBI and MPI-CorrBench. In particular, *none*, *vector*, and *index* refer to no normalization, normalization across the vector, or per element. The impact of optimizing normalization is less than 3% across all the scenarios.

Feature selection. As described in Section IV-A, features

Tool	Errors			Results				Robustness		Usefulness				Overall accuracy
	CE	TO	RE	TP	TN	FP	FN	Coverage	Conclusiveness	Specificity	Recall	Precision	F1 Score	
ITAC	0	157	1	859	738	4	102	1	0.915	**0.995**	0.894	**0.995**	**0.942**	0.858
PARCOACH V2.3.1	0	0	0	775	66	679	341	1	1	0.088	0.694	0.533	0.603	0.452
IR2vec Intra	0	0	0	1043	664	81	73	1	1	0.891	**0.935**	0.928	0.931	**0.917**
IR2vec Cross	0	0	0	805	523	311	222	1	1	0.627	0.784	0.721	0.751	0.714
GNN Intra	0	0	0	1045	657	88	71	1	1	0.902	0.922	0.936	0.929	0.853
GNN Cross	0	0	0	1108	19	726	8	1	1	0.703	0.604	0,993	0,751	0,830
Ideal tool	*0*	*0*	*0*	*1116*	*745*	*0*	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*

TABLE III

DETAILED METHODS EVALUATION AGAINST THE MPI BUGS INITIATIVE. BEST RESULTS ARE IN BOLD. ALL THE PREDICTIONS ARE ON WHETHER THE CODE IS CORRECT OR INCORRECT.

Compilation option	Normalization	Dataset	Results				Metrics			
			TP	TN	FP	FN	Recall	Precision	F1 Score	Accuracy
-O0			1042	681	64	74	0.934	0.942	0.938	0.926
-O2	none	MBI	1039	662	83	77	0.931	0.926	0.929	0.914
-Os			1047	665	80	69	0.938	0.929	0.934	0.920
-O0			205	191	11	9	0.9579	0.9491	0.9535	0.9519
-O2	none	CORR	198	180	22	16	0.9252	0.9000	0.9124	0.9087
-Os			203	189	13	11	0.9486	0.9398	0.9442	0.9423
-O0			1038	679	66	78	0.930	0.940	0.935	0.923
-O2	vector	MBI	1029	659	86	87	0.922	0.923	0.922	0.907
-Os			1043	664	81	73	0.935	0.928	0.931	0.917
-O0			202	186	16	12	0.9439	0.9266	0.9352	0.9327
-O2	vector	CORR	194	187	15	20	0.9065	0.9282	0.9173	0.9159
-Os			200	184	18	14	0.9346	0.9174	0.9259	0.9231
-O0			1042	681	64	74	0.934	0.942	0.938	0.926
-O2	index	MBI	1039	662	83	77	0.931	0.926	0.929	0.914
-Os			1047	665	80	69	0.938	0.929	0.934	0.920
-O0			205	188	14	9	0.9579	0.9361	0.9469	0.9447
-O2	index	CORR	202	188	14	12	0.9439	0.9352	0.9395	0.9375
-Os			203	189	13	11	0.9486	0.9398	0.9442	0.9423

TABLE IV

RESULTS FOR IR2VEC INTRA WITH DIFFERENT COMPILATION AND NORMALIZATION OPTIONS. CORR REFERS TO MPI-CORRBENCH. ALL THE PREDICTIONS ARE ON WHETHER THE CODE IS CORRECT OR INCORRECT.

have a significant impact on the prediction accuracy. In addition to normalizing the features, we applied a feature selection process where we only select a subset of the features to remove the noise with GA. Table V presents the benefit of running a feature selection with GA over naively using the vector. Results were obtained with the $-Os$ and *vector* as compilation and normalisation options, respectively. Interestingly, feature selection improves the accuracy by 5% and up to 47% for Intra and Cross respectively. This means that the Cross scenario is more sensitive to this parameter.

Seeds. We also noticed that the seed used in our model to generate the embedding in IR2vec can have a significant impact on the subsequent predictions. In particular, we performed a GA exploration to select relevant features over an original seed and then we re-generated vectors with IR2vec using a different seed. With *-Os* compilation and *vector* normalization, changing the seed of the embedding resulted in 0.6% losses and no losses for Intra MBI and MPI-CorrBench respectively.

Nevertheless, accuracy losses are expected since the GA has been trained for the original seed. This is particularly the case for Cross where the GA played a critical in the good predictions: we observed accuracy losses of 40.81% and 2.79% when validating over MPI-CorrBench and MBI respectively, indicating that the GA exploration must be adjusted to the embedding in some scenarios (i.e., predicting from MBI to MPI-CorrBench).

Prediction labels. So far, we trained models to determine if a code is correct or incorrect. However, we can further expand our predictions by determining the actual error type instead of just if the code is correct or incorrect. In particular, we trained our DT using the labels of the datasets to directly predict the error type. Please note that this approach is not possible for Cross as the error types are different between training and validation in that scenario. Figure 6 presents the prediction accuracy per label of our models over MBI.

We observe 3 large categories of prediction errors: accurately predicted labels with accuracy over 90% (e.g., Correct, Call Ordering, and Epoch Lifecycle), labels that are mostly correctly predicted (e.g., Invalid Parameter, Parameter Matching) with accuracies around 75%, and completely misspredicted labels such as Message Race or Resource Leak. To understand why our model miss predict some labels, we investigated the number of occurrences per code label in the dataset. Resource Leak has only 14 instances in our dataset, making it very difficult to learn for the model across the 10 verification folds. Interestingly, Message Race has more instances than Epoch Lifecycle (which is perfectly predicted): thus the number of samples is not the only reason for the

Model	GA	Dataset		Results				Metrics			
		Training	Validation	TP	TN	FP	FN	Recall	Precision	F1 Score	Accuracy
IR2vec Intra	OFF	MBI	MBI	989	635	110	127	0.886	0.900	0.893	0.873
	ON			1043	664	81	73	0.935	0.928	0.931	0.917
	OFF	CORR	CORR	190	174	28	24	0.888	0.871	0.879	0.875
	ON			200	184	18	14	0.935	0.917	0.926	0.923
IR2vec Cross	OFF	MBI	CORR	151	92	63	110	0.578	0.706	0.636	0.584
	ON			182	176	32	26	0.875	0.850	0.863	0.861
	OFF	CORR	MBI	955	235	161	510	0.652	0.856	0.740	0.639
	ON			805	523	311	222	0.784	0.721	0.751	0.714

TABLE V

RESULTS FOR IR2VEC INTRA AND CROSS WITH AND WITHOUT GA. CORR REFERS TO MPI-CORRBENCH. ALL THE PREDICTIONS ARE ON WHETHER THE CODE IS CORRECT OR INCORRECT.
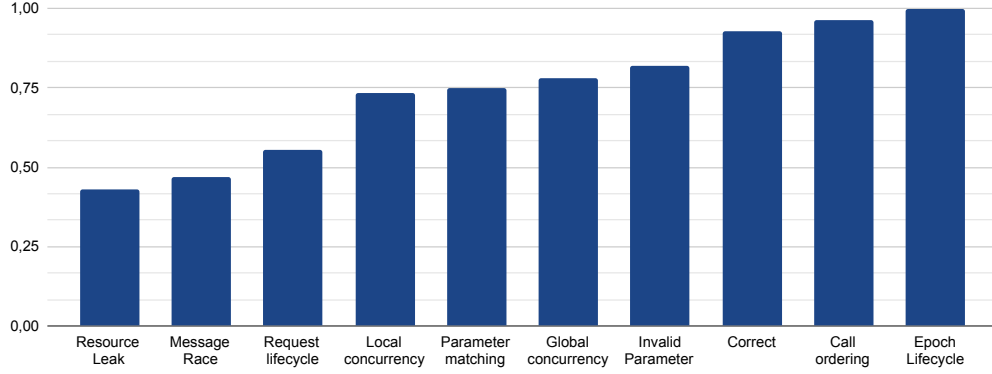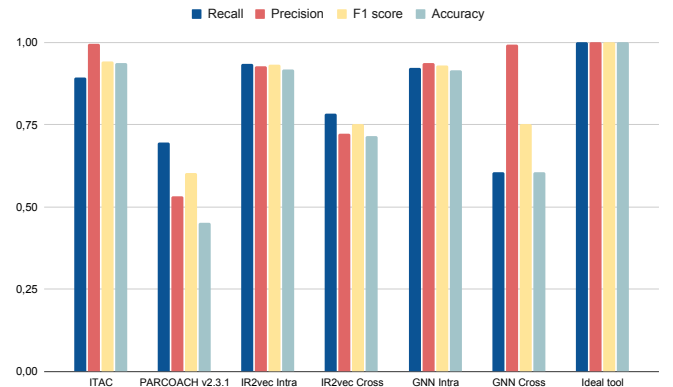

Fig. 6. Prediction accuracy of IR2vec per label. We trained a DT classifier with MBI to predict each label separately. Accuracy is calculated as the number of labels correctly predicted in the validation folds divided by the total number of codes with that label. The label prediction quality significantly depends on error types.
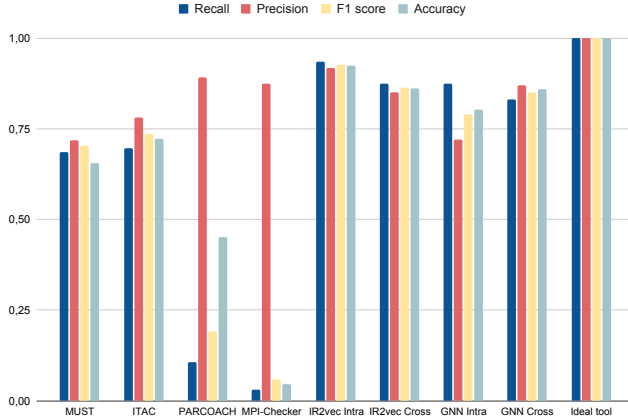
(a) MPI-CorrBench

(b) MBI

Fig. 7. Metrics Results on MPI-CorrBench (left) and MBI (right). For MPI-CorrBench, results of MUST, ITAC, PARCOACH and MPI-Checker are coming from [3].

miss-prediction. We suppose that errors exhibit different code patterns, and some are easier to identify with ML approaches. We further investigate the interaction between the errors in Section V-E.

End results. As shown in Table II, IR2vec outperforms GNN with a recall of 0.935, a precision of 0.928 and a F1 score o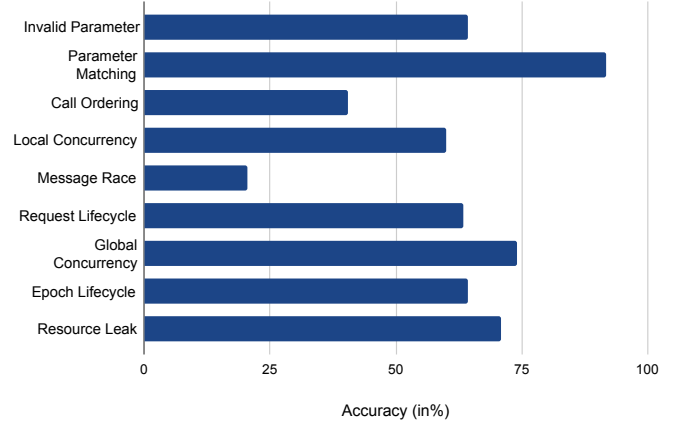f 0.931. Both models show better results on MBI compared to MPI-CorrBench. This may be explained by the higher number of codes in MBI: the models have more codes in the training set.

B. Mix Modeling

We further extended our models to operate over the dataset Mix composed of both MBI and MPI-CorrBench. The goal of
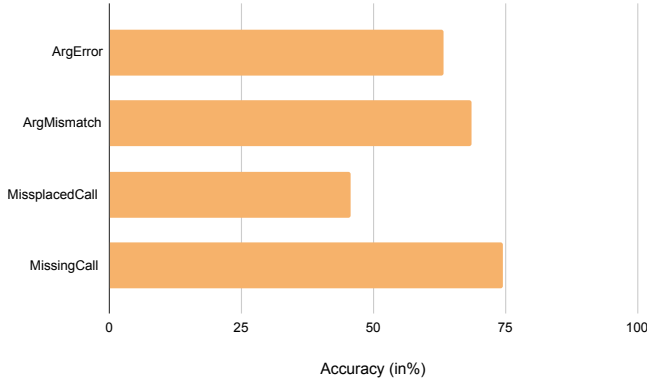
(a) MPI-CorrBench

(b) MBI

Fig. 8. Ablation study results for MPI-CorrBench (left) and MBI (right).

this scenario is to demonstrate if models can predict errors in a larger and more diverse context. For consistency, we again employed a 10-fold cross-validation. The outcomes from mix training and validation is also presented in Table II (lines *IR2vec Mix* and *GNN Mix*) and mirrors the results achieved in the Intra dataset tests. Specifically, we achieved promising results across various metrics: a recall and F1 score of 0.893 and an accuracy of 0.911 for the GNN model. These numbers, being in close alignment with the GNN Intra results, highlight the robustness and reliability of our model. Interestingly IR2vec slightly decreases to 0.882. It is possible that the GNN method can easily scale to larger datasets.

C. Cross Modeling

Finally, we trained and validated our methods over distinct datasets. This experiment tests the model's ability to generalize and detect errors in untrained, unseen data. This is important because it is desirable that a model does not only rely on the specific error types it was trained on, but to also be able to detect new error types based on its understanding of the underlying code patterns. Results are referred as *IR2vec Cross* and *GNN Cross* in table II.

GNN models struggled to generalize their learnings across different datasets. This is particularly notable when using MPI-CorrBench as a training dataset and MBI as a validation dataset. We obtained an accuracy score of 0.605 with the GNN model. The insights and patterns that the model learned from MPI-CorrBench do not seamlessly translate to MBI. Such observation raises pertinent questions about the transferability of knowledge in models, emphasizing the need for further refinement to enhance their cross-benchmark applicability.

Nevertheless, we note that IR2vec achieved an accuracy of 0.713 and 0.86 for MPI-CorrBench and MBI validation respectively. These numbers are promising for such difficult scenarios as both code structures and errors labels are different across the datasets. The GA feature selection was critical in

achieving them. Indeed, while feature selection moderately impacts Intra predictions, it has a significant impact when cross predicting. In particular, we observed that feature selection improved the accuracy by 47% and 12% when predicting MPI-CorrBench and MBI respectively.

D. Comparison with Related Works

Figure 7 shows the Recall, Precision, F1 score, and Accuracy results of several state of the art verification tools on MPI-CorrBench and MBI. On both subfigures, the last bars depict the results of an ideal tool.

We investigate MPI-CorrBench in Figure 7 (a). We used the results presented in [3] for MUST, ITAC, PARCOACH and MPI-Checker. We compared these results against the prediction of our different models (i.e., IR2vec and GNN) in different scenarios (i.e., Intra and Cross). Our methods outperform the existing verification tools or at least achieve similar results in the most restrictive scenario cross. Our methods achieve a score of at least 0.75. Furthermore, IR2vec Intra has the closest results to an ideal tool.

Figure 7 (b) presents the results over MBI. Because we used a different version of MBI than in [2], we had to reproduce the experiments for PARCOACH and ITAC. We also used the last versions of these tools. We chose to compare our method only with ITAC and PARCOACH as ITAC is the best tool in [2] and PARCOACH is the only static tool used in MBI. *This makes a fair comparison as our approach is also static* (as we also only study the LLVM IR). We observe that ITAC has the best precision, F1 score and accuracy. Yet, IR2vec Intra shows competitive results to ITAC, and more importantly, does not require executing the applications. Indeed, static analyses enable an early detection of errors and avoid the cost of program execution. Therefore, our method can easily be integrated into an automatic toolchain where, at compilation, a light ML-based verification step checks the code.

Table III further details MBI results. It gives the number of compilation errors (CE), time out (TO), runtime errors (RE),

TP, TN, FP and FN as well as seven metrics, defined in [2], depicting the robustness, the usefulness and overall accuracy of the different methods. The last row displays the results of an ideal tool and best results are in bold. The tools and our models have all a coverage of 1 as none of them have compilation error. However, ITAC has 157 time out and 1 runtime error which leads to a conclusiveness (i.e., ability to draw a diagnostic on codes) of 0.915. ITAC has the best specificity, precision and F1 score whereas IR2vec Intra has the best recall and overall accuracy (and unlike ITAC, operates statically). We further compare the different approaches in Section VI.

E. Ablation Study

The goal of this subsection is to study the interaction between the different error labels. The idea is to remove one labelled error from all the training sets and evaluate if the resulting models can detect it in the validation sets. To do so, we reproduced the 10 folds cross validation but in addition ensured that no samples of the target label are ever present in the training codes. The model capabilities to predict erroneous codes that it has never seen before demonstrate 1) its generalization to new scenarios as well as 2) the code patterns that are shared between the different error labels. We implemented this study with IR2vec over the datasets presented in Section V-A.

Model generalization. Figure 8 presents results of the ablation study for both MPI-CorrBench and MBI. Each bar represents the prediction accuracy per label (i.e., when the label is excluded from training and only occurs at validation). For each label, we calculate the accuracy as the number of samples of the label correctly predicted as incorrect divided by the total count of that label. Note that we trained the model to predict correct or incorrect and not the label itself as it does not appear in the training codes.

Labels such as Parameter Matching, MissingCall, or Global Concurrency have a high accuracy score (around or over 75%). On the opposite some labels such as Message Race or MissplacedCall are very difficult to generalize over. It is interesting to note that Resource Leak is better predicted in the ablation study than when we directly train models to identify it (see figure 6 for prediction per label). The ablation model determines if a code is correct or incorrect while the Intra model explicitly predicts it: it is therefore likely that our Intra model confused it with some other related error. Overall, our model has the potential to be applied on new errors that it has not encountered before.

Error interaction. Figure 9 shows the prediction accuracy when two labels are excluded from training with MPI-CorrBench as a dataset. While MissingCall was well predicted when excluded from the training set, its accuracy score falls down to 44% when ArgError is also excluded from training. This shows a similarity between the two error types (they exhibit differences at source but similar embeddings help detect them). Conversely, MissplacedCall has a higher accuracy score if ArrgError is not in training. For MBI, Parameter Matching

decreases from 92% to 77% when it is excluded with Resource Leak. Our model is unable to detect Epoch Lifecycle (accuracy of 0) if Parameter Matching, Call Ordering or Message Race is also removed from training. Like MissplacedCall, Message Race has a higher accuracy if Parameter Matching is removed from training.

In other words, these prediction scores can be used to quantify how much two errors share the same code patterns: MissingCall from MPI-CorrBench has similar code patterns as ArgError. We believe such metrics have potential to guide the errors topology definition.
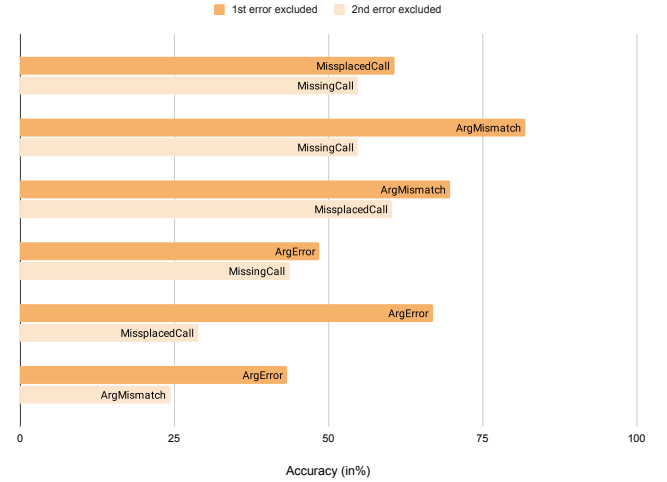

Fig. 9. Ablation study results for MPI-CorrBench when two labels are excluded from training.

F. Preliminary Real Case Scenario

A limitation of our approach is the scale of the experiments. In our knowledge, MPI-CorrBench and MBI are the only two benchmarks with correct and incorrect MPI codes. We can use mutation techniques or GitHub to acquire new incorrect cases, but we decided to start with these existing correctness benchmarks as they are the standards for evaluating active verification tools, enabling a fair comparison. Large-scale exploration is a promising future direction for us that we discuss in Section VI.

To try our models beyond benchmarks, on a real case, we investigate Hypre, a library of high performance preconditioners and solvers featuring multigrid methods, available on Github [1]. An error due to the use of the same tag in two MPI operations is fixed on commit bc3158e (version 2.10.1). We retrieved the code before and after this commit in order to have a correct and incorrect version of the code. Both versions of the code can be used by our models to evaluate if the cross modeling has the potential to detect errors in real world applications.

To extract the features, we start by compiling the codes. As described previously, we consider the tree compiler options:

---

[1]https://github.com/hypre-space/hypre

Training Dataset	Features	Compilation option - code correct/incorrect					
		O0-ok	*O2-ok*	*Os-ok*	*O0-ko*	*O2-ko*	*Os-ko*
MBI	all	ok	ok	ok	ok	ok	ok
MPI-CorrBench	all	ok	ok	ok	ok	ok	ok
MBI	GA	ko	ok	ok	ko	ko	ko
MPI-CorrBench	GA	ko	ok	ok	ko	ko	ko

TABLE VI

PREDICTION ON HYPRE USING MODELS TRAINED ON EITHER MBI OR MPI-CORRBENCH. *ok* REFERS TO CORRECT CODES WHILE *ko* REFERS TO INCORRECT CODES. EACH COLUMN ON THE RIGHT SIDE REPRESENTS A VERSION OF HYPRE (EITHER CORRECT OR INCORRECT) COMPILED WITH *-O0*, *-O2*, OR *-Os*. EACH LINE REPRESENTS A MODEL TRAINED USING THE DATASET ALONG WITH THE DESCRIBED FEATURES. THE VALUE OF THE CELL INDICATES THE MODEL PREDICTION ON THE COLUMN CODE: OK OR KO FOR CORRECT OR INCORRECT, RESPECTIVELY. THE CELL COLOR SHOWS IF THE MODEL CORRECTLY PREDICTED (IN GREEN) THE CODE LABEL OR IF IT MADE AN ERROR (RED).

-O0, *-O2*, and *-Os*. Each resulting IR is used by IR2Vec to generate vectors representing the codes. The vectors are subsequently normalized with *vector* as in the *IR2Vec Cross* evaluation. We trained our models on either MBI or MPI-CorrBench.

Table VI presents the predictions of the different models. We evaluated the models without feature selection (*all*), or with GA (*GA*) following the same procedure as in Section V-A. Without features selection, our models fail to detect the error in Hypre (red parts in the first two lines). However, when we select different features, the models successfully label the code (and correctly predict the *cross* scenario), independently if they were trained on MBI or MPI-CorrBench. We note that by changing the features, *O0-ok* can accurately be predicted as correct, but that we did not find any combination of features that successfully label all Hypre versions. This further indicates that the compiler optimization used to generate the code representation must be considered with the machine learning models as both impact the predictions.

VI. DISCUSSION AND FUTURE WORK

Detecting errors in MPI programs is challenging, and no method is able to detect all kinds of errors. Expert tools and ML-based approaches try to address this challenge and seem to achieve similar results. Yet, they employ drastically different approaches. Expert tools require human expertise to identify and manually devise algorithms and heuristics implemented in the tool for diverse error patterns. While costly in manpower and time, such approaches could provide the benefit of understanding why a code is assumed incorrect. Conversely, with ML methods, currently, we cannot easily understand why a code is predicted as incorrect (or ensure that a predicted code is correct for the errors that our model predicts). This is a limitation of ML strategies, which is now an important research direction in explainable AI research. Nevertheless, ML methods only require a new dataset to consider new bug issues. Thus, we expect such methods to easily generalize over new emerging scenarios.

Indeed, and as future work, we plan to apply our models on larger scales. By crawling GitHub repositories, we can use our models as detectors to identify bugs in existing MPI projects. We can also take the GitHub codes as additional training datasets for our models. The main challenge will be the labeling of the data. Our insights are to track how specific errors impact code embedding or look at the GitHub metadata. We also consider training models to predict the error location. A first step in that direction is applying our models at different code granularities by extracting the code into different compilation units. Whether or not an error is detected across the different compilation units can serve as a guideline for the exact error location and what caused it. Finally, we envision training Large Language Models to propose code fixes directly.

VII. CONCLUSION

To the best of our knowledge, this paper presents the first method using machine learning techniques to detect errors in MPI programs. We developed two models that either use embedding or deep learning graph neural networks. We trained and validated these models on three datasets and compared them with existing MPI verification tools. The ML methods achieved competitive results across the datasets compared to the expert tools. Furthermore, while our models do not provide feedback like expert specialized tools, they show promising generalization capabilities over new unseen error types or benchmark suites. These results show the potential of ML-based approaches in the context of MPI verification.

[2]https://researchit.las.iastate.edu
[3]https://www.plafrim.fr
[4]https://www.grid5000.fr

REFERENCES

[1] G. Gopalakrishnan, P. D. Hovland, C. Iancu, S. Krishnamoorthy, I. Laguna, R. A. Lethin, K. Sen, S. F. Siegel, and A. Solar-Lezama, "Report of the HPC correctness summit, jan 25-26, 2017, washington, DC," *CoRR*, vol. abs/1705.07478, 2017. [Online]. Available: http://arxiv.org/abs/1705.07478

[2] M. Laurent, E. Saillard, and M. Quinson, "The mpi bugs initiative: a framework for mpi verification tools evaluation," in *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*, 2021, pp. 1–9.

[3] J.-P. Lehr, T. Jammer, and C. Bischof, "Mpi-corrbench: Towards an mpi correctness benchmark suite," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 69–80. [Online]. Available: https://doi.org/10.1145/3431379.3460652

[4] A. Droste, M. Kuhn, and T. Ludwig, "Mpi-checker: Static analysis for mpi," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2833157.2833159

[5] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers, "Civl: the concurrency intermediate verification language," in *SC*, Nov 2015, pp. 1–12.

[6] Z. Chen, H. Yu, X. Fu, and J. Wang, "Mpi-sv: a symbolic verifier for mpi programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 93–96. [Online]. Available: https://doi.org/10.1145/3377812.3382144

[7] D. Khanna, S. Sharma, C. Rodríguez, and R. Purandare, "Dynamic symbolic verification of mpi programs," in *Formal Methods*, K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, Eds. Cham: Springer International Publishing, 2018, pp. 466–484.

[8] K. Sen, D. Marinov, and G. Agha, "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*, ser. ESEC/FSE-13. New York, NY, USA: Association for Computing Machinery, 2005, p. 263–272. [Online]. Available: https://doi.org/10.1145/1081706.1081750

[9] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 213–223. [Online]. Available: https://doi.org/10.1145/1065010.1065036

[10] H. Li, S. Li, Z. Benavides, Z. Chen, and R. Gupta, "Compi: Concolic testing for mpi applications," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 865–874.

[11] H. Li, Z. Chen, and R. Gupta, "Efficient concolic testing of mpi applications," in *Proceedings of the 28th International Conference on Compiler Construction*, ser. CC 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 193–204. [Online]. Available: https://doi.org/10.1145/3302516.3307353

[12] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014. [Online]. Available: http://hal.inria.fr/hal-01017319

[13] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, "Isp: a tool for model checking mpi programs," in *PPOPP*, 2008.

[14] S. Böhm, O. Meca, and P. Jančar, "State-space reduction of non-deterministically synchronizing systems applicable to deadlock detection in mpi," in *FM 2016: Formal Methods*, J. Fitzgerald, C. Heitmeyer, S. Gnesi, and A. Philippou, Eds. Cham: Springer International Publishing, 2016, pp. 102–118.

[15] T. Hilbrich, M. Weber, J. Protze, B. R. de Supinski, and W. E. Nagel, "Runtime correctness analysis of mpi-3 nonblocking collectives," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016. New York, NY, USA: ACM, 2016, pp. 188–197. [Online]. Available: http://doi.acm.org/10.1145/2966884.2966906

[16] T. Hilbrich, M. S. Müller, B. R. de Supinski, M. Schulz, and W. E. Nagel, "Gti: A generic tools infrastructure for event-based tools in parallel systems," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 1364–1375.

[17] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. Supinski, M. Schulz, and G. Bronevetsky, "A scalable and distributed dynamic formal verifier for mpi programs," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–10.

[18] "Intel trace analyzer and collector," https://software.intel.com/-content/www/us/en/develop/tools/oneapi/components/trace-analyzer.html.

[19] T.-D. Diep, K. Fürlinger, and N. Thoai, "Mc-cchecker: A clock-based approach to detect memory consistency errors in mpi one-sided applications," in *EuroMPI'18*, 2018.

[20] Z. Chen, J. Dinan, Z. Tang, P. Balaji, H. Zhong, J. Wei, T. Huang, and F. Qin, "Mc-checker: Detecting memory consistency errors in mpi one-sided applications," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 499–510.

[21] C. Falzone, A. Chan, E. Lusk, and W. Gropp, "Collective error detection for mpi collective operations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, B. Di Martino, D. Kranzlmüller, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 138–147.

[22] J. L. Träff and J. Worringen, "Verifying collective mpi calls," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 18–27.

[23] E. Saillard, P. Carribault, and D. Barthou, "PARCOACH: combining static and dynamic validation of MPI collective communications," *IJHPCA*, vol. 28, no. 4, pp. 425–434, 2014.

[24] P. Huchant, E. Saillard, D. Barthou, H. Brunie, and P. Carribault, "PARCOACH Extension for a Full-Interprocedural Collectives Verification," in *Second International Workshop on Software Correctness for HPC Applications*, 2018.

[25] V. M. Nguyen, E. Saillard, J. Jaeger, D. Barthou, and P. Carribault, "Parcoach extension for static mpi nonblocking and persistent communication validation," in *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*, 2020, pp. 31–39.

[26] T. C. Aitkaci, M. Sergent, E. Saillard, D. Barthou, and G. Papauré, "Dynamic Data Race Detection for MPI-RMA Programs," in *EuroMPI 2021 - European MPI Users's Group Meeting*, Munich, Germany, Sep. 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03374614

[27] E. Saillard, M. Sergent, T. C. Aitkaci, and D. Barthou, "Static Local Concurrency Errors Detection in MPI-RMA Programs," in *Correctness 2022 - Sixth International Workshop on Software Correctness for HPC Applications*, Dallas, United States, Nov. 2022. [Online]. Available: https://hal.inria.fr/hal-03882459

[28] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[29] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[30] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," *Advances in Neural Information Processing Systems*, vol. 34, pp. 27 865–27 876, 2021.

[31] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations (ICLR)*, 2020.

[32] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[33] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 479–490.

[34] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 87–98.

[35] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, 2016.

[36] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *IJCAI*, 2017, pp. 3034–3040.

[37] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. Srikant, "Ir2vec: Llvm ir based scalable program embeddings," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–27, 2020.

[38] N. Ayewah and W. Pugh, "The google findbugs fixit," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 241–252.

[39] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[40] D. A. Ramos and D. Engler, "{Under-Constrained} symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 49–64.

[41] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[42] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2019.

[43] G. Lin, J. Zhang, W. Luo, L. Pan, O. De Vel, P. Montague, and Y. Xiang, "Software vulnerability discovery via learning multi-domain knowledge bases," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2469–2485, 2019.

[44] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. O'Boyle, and H. Leather, "Programl: A graph-based program representation for data flow analysis and compiler optimizations," in *International Conference on Machine Learning*. PMLR, 2021, pp. 2244–2253.

[45] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[46] "Pyeasyga," https://pyeasyga.readthedocs.io/en/latest/readme.html, accessed: 2022-07-29.

[47] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[48] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" *arXiv preprint arXiv:2105.14491*, 2021.

[49] A. TehraniJamsaz, M. Popov, A. Dutta, E. Saillard, and A. Jannesari, "Learning intermediate representations using graph neural networks for numa and prefetchers optimization," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2022, pp. 1206–1216. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPS53621.2022.00120

[50] Z. Li, P. Ma, H. Wang, S. Wang, Q. Tang, S. Nie, and S. Wu, "Unleashing the power of compiler intermediate representation to enhance neural program embeddings," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2253–2265.