THESIS

# TOWARDS HETEROGENEITY-AWARE AUTOMATIC OPTIMIZATION OF TIME-CRITICAL SYSTEMS VIA GRAPH MACHINE LEARNING

Submitted by

Ronaldo Armando Canizales Turcios

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2024

Master's Committee:

Advisor: Jedidiah McClurg

Sanjay Rajopadhye
Sudeep Pasricha

# ABSTRACT

## TOWARDS HETEROGENEITY-AWARE AUTOMATIC OPTIMIZATION OF TIME-CRITICAL SYSTEMS VIA GRAPH MACHINE LEARNING

Modern computing's hardware architecture is increasingly heterogeneous, making optimization challenging; particularly on time-critical systems where correct results are as important as low execution time. First, we explore a study case about the manual optimization of an earthquake engineering-related application, where we parallelized accelerographic records processing. Second, we present egg-no-graph, our novel code-to-graph representation based on equality saturation, which outperforms state-of-the-art methods at estimating execution time. Third, we show how our 150M+ instances heterogeneity-aware dataset was built. Lastly, we redesign a graph-level embedding algorithm, making it converge orders of magnitude faster while maintaining similar accuracy than state-of-the-art on our downstream task, thus being feasible for use on time-critical systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF CODE LISTINGS

# Chapter 1

# Introduction

Modern computing is becoming increasingly heterogeneous in terms of hardware architecture, making it challenging to optimize systems both manually and automatically. In time-critical systems, correctness not only depends on the results obtained but also on the time at which they are executed. The goals of this work are to study manual optimizations of time-critical systems and develop a heterogeneity-aware tool for performance characterization via graph machine learning.

This document is structured as follows: Chapter 1 states the research questions and contributions of the two projects we have developed; Chapter 2 is the study case about manual optimization of a time-critical system related to seismology engineering where accelerographic records are processed in parallel using OpenMP, Chapter 3 presents how we (a) designed a novel code-to-graph representation based on e-graphs (equality saturation), offering more fine-grained graphs without exploding in size, alongside the example that motivated it, (b) redesigned a graph-level embedding algorithm with comparable accuracy to 6 state-of-the-art approaches while executing orders of magnitude faster, (c) created a heterogeneity-aware dataset with 150M+ training instances including 13 hardware architectures, and (d) obtained experimental results by training several models, achieving NRMSEs around 4%, 0.56% and 0.87% under different combinations of granularity and vector lengths. Lastly, Chapter 4 concludes with final thoughts about this and future work.

## 1.1 Research questions

Regarding our study case of manual optimization of a time-critical system:

- **RQ1.** How can we manually optimize a sequential application, implemented in a mix of Fortran and C++, for strong-motion processing currently used in a Salvadoran National Lab through OpenMP pragmas? How much raw speedup can be obtained, and how does it correlate with problem size (defined as data points distributed among different seismic stations)?

Regarding the performance characterization via Graph Machine Learning:

- **RQ2.** How can a traditional code-to-graph representation such as SSAs be augmented using equality saturation to improve their representational capacity for programs written in C/C++?
- **RQ3.** Can a traditional graph embedding method such as WL color refinement be improved to be both suitable for machine learning and efficient enough for use in time-critical systems? Does it provide a reasonable tradeoff between speed and accuracy?
- **RQ4.** How can we design a heterogeneity-aware dataset suitable for graph machine learning that includes various hardware architectures, a variety of time-sensitive programs, workloads (resource availability), problem sizes, and multiple target performance metrics?
- **RQ5.** Using RQ4's dataset, does a coarse vs. a fine-grained training approach report the lowest end-to-end NRMSE for the downstream task of predicting execution time?
- **RQ6.** Using RQ5's best training approach, how do end-to-end NRMSE and training time behave among different embedding vector lengths for egg-no-graph, RQ2?

## 1.2 Contributions

1. **A fully-parallelized approach for strong-motion record processing.** Leveraging parallel loops and task parallelization, our method successfully addresses the challenge of efficiently processing accelerographic data, providing scalability and speedup roughly proportional to problem size. Through experimentation with more than one million data points from six real-world seismic events, our approach achieved speedups of up to 2.9x over the baseline. Our implementation utilized OpenMP on both Fortran and C++.
2. **Egg-no-graph: Design of a code graph representation.** We designed a code-to-graph algorithm using SSAs as the backbone and converting them to an e-graph. Leveraging the idea of equality saturation in these graphs increased their expressive power, allowing them

to uniquely represent more programs than originally possible with traditional graphs such as ASTs and SSAs. Thus increasing the breadth of programs that can be uniquely modeled.

3. **Easy Egg: Python implementation using EasyGraph and Egglog.** We created a fast converter, 4.35ms on average, for the code-to-graph method mentioned above, leveraging widely used open-source libraries. EasyGraph contains several state-of-the-art graph embedding algorithms, while Egglog is a Python interface of a robust Rust e-graph toolkit.

4. **Bag of Colors: Implementation of fast WL-based graph embedding algorithms.** We modified a color refinement-based graph embedding algorithm inspired by the Weisfeiler-Lehman graph-isomorphism test. By preventing the phenomenon known as over-smoothing, our version is not only more suitable for usage in ML than the vanilla "1-WL" but also 20x up to 5000x faster (0.85ms on average) compared to six state-of-the-art algorithms, while maintaining comparable accuracy. Thus making it suitable for use in time-critical systems.

5. **Design and build a 150M+ instances heterogeneity-aware training dataset.** Targeting three High-Performance Computing (HPC) and Competitive Programming (CP) famous repositories, we considered a design space of 13 unique hardware architectures, 12 problem sizes, 8 workloads, 6 performance metrics, 5 code-to-graph representations, 7 embedding algorithms, and 8 vector length sizes. Its computing time required around 10 weeks.

6. **Experimental results: coarse vs fine-grained training.** We compared the NRMSE of (a) six algorithms trained directly over a whole dataset vs. (b) a hierarchical approach where the same dataset is divided into clusters, and individual cluster-wise regressors are trained and finally composed in a single ensemble model. While approach (a) achieved 4% NRMSE, (b) reached values $<1\%$. In both scenarios, Eggnog outperformed state-of-the-art approaches.

7. **Experimental results: Eggnog's performance.** We explore the behavior of our method on the (b) approach detailed above across all programs in our dataset. We report trends for end-to-end NRMSE and fine-grained training times for four embedding vector lengths.

# Chapter 2

# Study case: Manual optimization of Accelerographic data processing

This chapter is about the paper "Parallelizing Accelerographic Records Processing" which I first authored and published at ParSocial'24 (IEEE Workshop on Parallel and Distributed Processing for Computational Social Systems) co-located with IPDPS'24 (38th IEEE International Parallel & Distributed Processing Symposium). The abstract of this paper is: Strong-motion processing holds paramount importance in earthquake engineering and disaster risk management systems. By leveraging parallel loops and task-parallelism techniques, we address computational challenges posed by large-scale accelerographic datasets. Through experimentation with more than one million data points from six real-world seismic events, our approach achieved speedups of up to 2.9x, demonstrating the effectiveness of parallel programming in accelerating seismic data processing. Our findings highlight the significance of parallel programming techniques in advancing seismological research and enhancing earthquake mitigation strategies.

## 2.1 Introduction

Earthquakes are well-known for their unpredictability and potential to cause significant damage. These events have profound and far-reaching impacts on both human societies and the environment. One of the most devastating earthquakes in recent history occurred in 2004 in Sumatra, Indonesia, triggering a massive tsunami that affected the entire Indian Ocean region, and resulting in approximately 230,000 fatalities [1]. The 2015 Nepal earthquake caused significant damage to the Himalayan region, with economic losses amounting to one-third of its gross domestic product for that year [2]. Similarly, in February 2023, a powerful earthquake struck southeastern Turkey and Syria, causing over 50,000 fatalities and widespread destruction [3]. Furthermore, seismic events such as the Gyeongju and Pohang earthquakes in 2016 and 2017 in the Korean Peninsula have underscored the importance of *ground-motion data collection and analysis* for ensuring the

stability of critical infrastructure such as nuclear power plants [4], and have highlighted the critical need for seismic hazard assessment and disaster risk reduction strategies in earthquake-prone areas, such as the Indian subcontinent [5].

Earthquake-resistant structural design is primarily concerned with the tradeoff between potential for local earthquake activity levels and the ability of structures to resist damage, and relies on seismic data collection and analysis. With numerous data analysis schemes available, ensuring accurate interpretation and utilization of the data is crucial [6]. Large seismic monitoring networks generate vast amounts of data that require efficient archiving, dissemination, visualization, and real-time analysis [7]. These records, obtained from *strong-motion accelerograph machines*, are indispensable for hazard estimation and site-effect studies, forming the backbone of seismic research [5]. *Processing strong ground motion data efficiently can mitigate seismic hazards and gain valuable insights into site-specific effects [8].*

In this paper, we explore a parallelization-based approach for improving the performance of vital strong-motion processing software used in El Salvador. Although various techniques for parallelizing generic programs exist, these approaches are not straightforward to use in the context of real-world seismological software because of complex process dependencies, legacy programming languages and APIs, mixed input/output data formats, and intrinsically distributed data sources.

Specifically, we focus on software used in El Salvador's Observatory of Natural Threats. In this context, despite using modern technology for sensing, distributing, storing, and visualizing seismic-related data, the core calculations needed for processing strong-motion records are performed using legacy Fortran code. In this paper, we show how to perform a careful analysis of input/output data dependencies, and apply techniques such as parallelizing loops and adding task-parallelism, producing an optimized version of the existing sequential implementation.

To our knowledge, our approach is the first to fully parallelize the processing of accelerographic records when all three signal components are analyzed, leveraging the robust Salvadoran strong-motion sensor network. We demonstrate the utility of our approach by processing data from 6 strong-motion events in El Salvador. On this data, our work outperforms the original sequential implementation by a factor of 2.6x to 2.9x, benefiting seismology research and natural disaster management, and aiding impact mitigation on society.

## 2.2 Background: Accelerographic Records Processing

When a seismic event occurs, proximal strong-motion sensors register its effects – Figure 2.1 shows an example of such sensors. Each sensor produces data files capturing three distinct components: longitudinal, transversal, and vertical motions. These raw data files, denoted as uncorrected versions, are stored with a *V1* extension, and contain the ground's acceleration, velocity, and displacement over a defined temporal window. The number of files needing to be processed per seismic event depends on nearby sensors' availability and the event's magnitude. In the following, we will outline the processing steps for each file.


**Figure 2.1:** Seismic station and accelerograph [9, 10]

The components of the accelerographic data (e.g., Figure 2.2) are individually stored in files named according to the format $[station][comp]$.`v1`. Subsequently, a Hamming band-pass filter is applied to each component, utilizing default parameters. This processing step generates corrected versions of the signals, which are then saved with a *V2* extension.

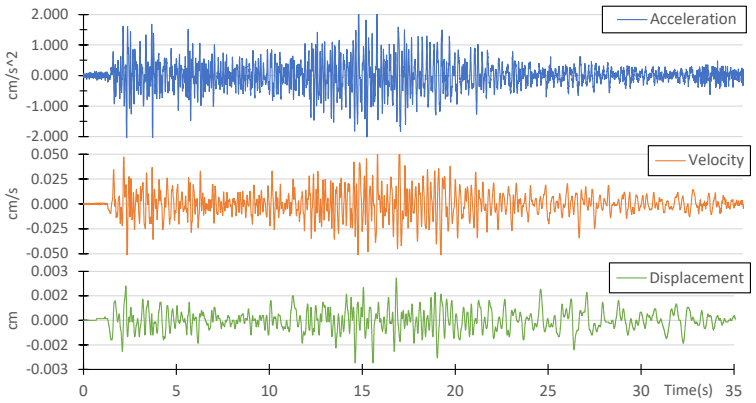



**Figure 2.2:** Accelerographic data (one component)

Following this correction, a Fourier transformation is performed on each signal (e.g, Figure 2.3) resulting in files marked with an *F* extension. Notably, the velocity Fourier spectrum of each signal holds significant importance, as its analysis yields the low-pass frequency (FPL) and low-stop frequency (FSL) parameters, highlighted in red in Figure 2.3. These are used in the definitive acceleration baseline correction of each signal. Moreover, the peak ground acceleration (PGA) values are extracted and archived.

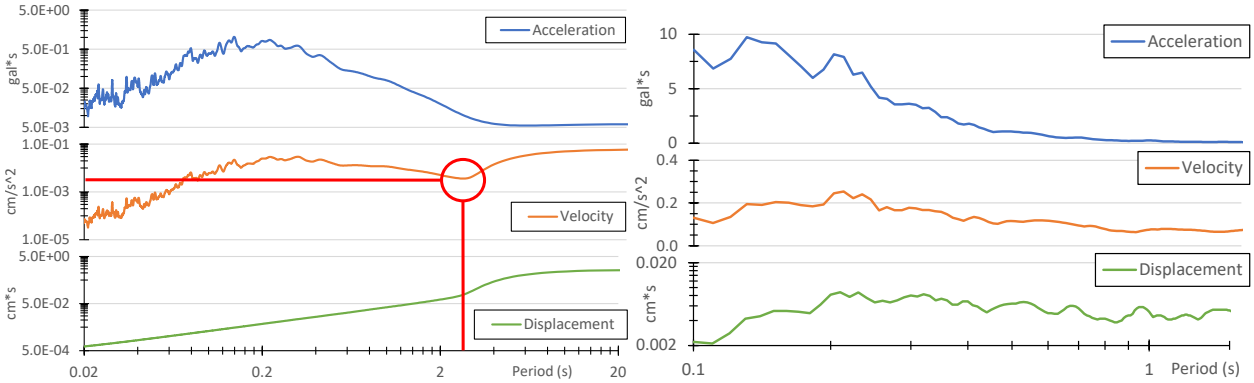



**Figure 2.3:** Left: Inflection point in the velocity Fourier spectrum, indicates the values of the FPL & FSL signal filtering parameters. Right: Response spectrum (both plots include only one component).

The final corrected signals are derived by applying another Hamming band-pass filter, tailored with the appropriate FPL and FSL parameters obtained from the Fourier analysis, and are subsequently stored in *V2* files.

The most computationally intensive operation involves calculating the Response spectrum for each corrected signal (e.g., Figure 2.3), saved with an *R* extension. This spectrum provides valuable insights into the response characteristics of various building types during seismic events.

Finally, 18 Global Earthquake Model (*GEM*) files are created from the *V2* and *R* files. These serve as crucial inputs for subsequent processes. This information resulting from this processing of strong-motion files is of considerable significance to structural engineers during the design phase of new buildings, informing their decisions and enhancing structural integrity.

## 2.3 Sequential Implementation

The entire process used in the sequential version is segmented into 20 sequential steps, shown in Figure 2.4. Each step, which we will refer to as a *process*, is either a function embedded within C++ code, or an entire Fortran program. While certain processes are lightweight, others entail substantial input/output operations, calculations, or plotting tasks.

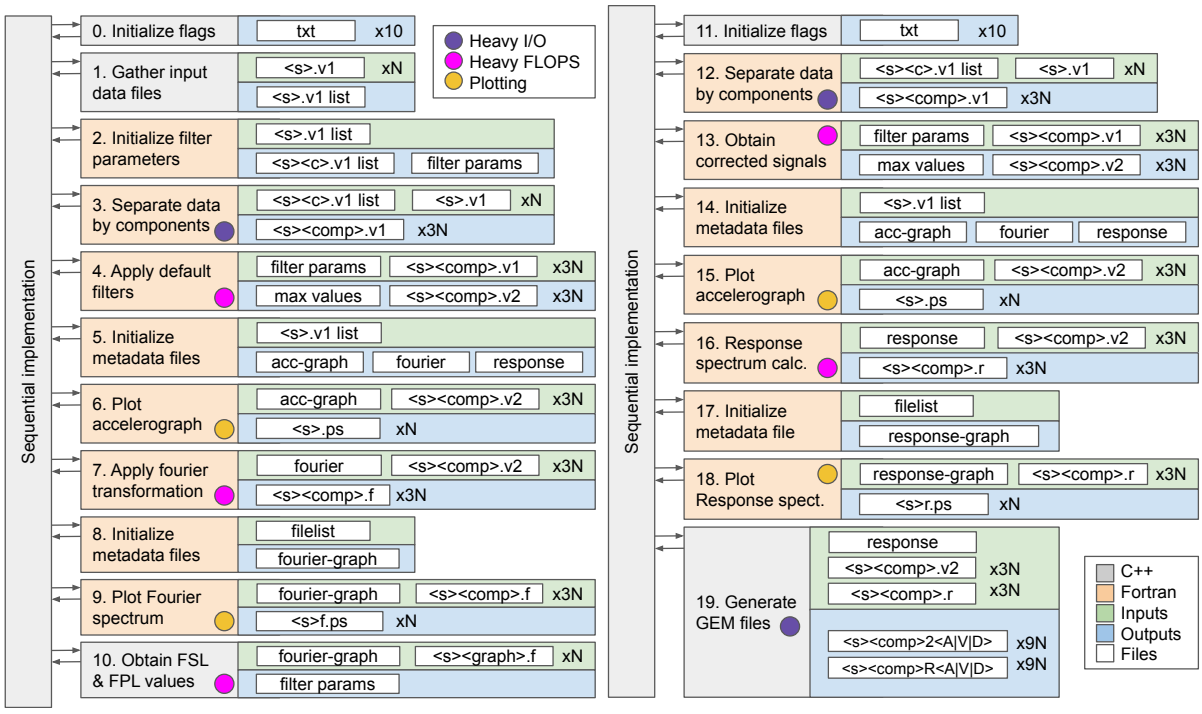



**Figure 2.4:** Sequential accelerographic records processing implementation

## 2.4 Optimizing the Sequential Implementation

Initial optimization involves the elimination of unnecessary processes. Detailed analysis of the process uncovered the following redundancies.

1. The plotting of uncorrected signals (P#6) is unnecessary and not utilized in this program. Additionally, the $[station]$.ps files produced are subsequently overwritten in P#15.

2. Segmentation of each component of uncorrected signals into individual files (P#12) is superfluous, since no modifications are applied to *V1* files during execution.

3. Overwriting intermediate files (P#14) is redundant; some data is not modified after P#5.

In summary, the exclusion of processes #6, #12, and #14 is feasible, and has no impact on the final output. This optimization reduces the overall execution time and avoids overwriting data. The resulting optimized sequential version is shown in Figure 2.5.

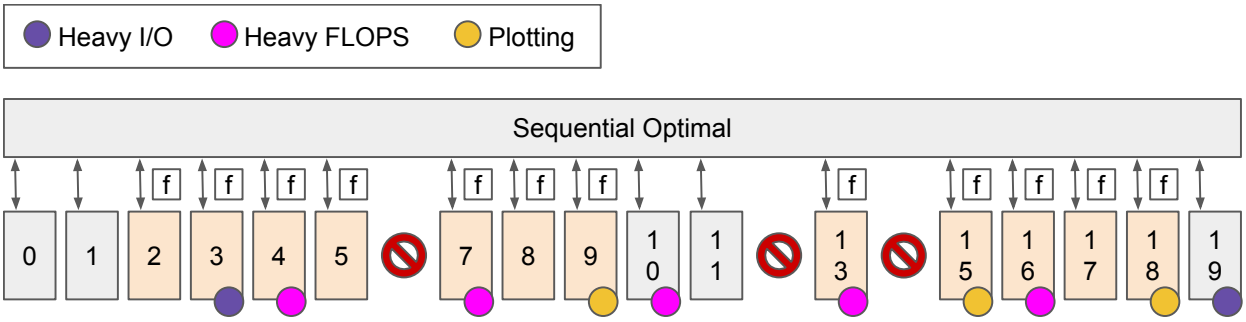



**Figure 2.5:** Optimized sequential version: processes overview

## 2.5 Partially Parallelized Implementation

Establishing the foundation for parallelizing the workflow involves delineating all processes and their dependencies, encompassing both inputs and outputs, as shown in Figure 2.6. These processes have been categorized into nine stages, with each stage slated for parallelization employing the specific strategies outlined on the right-hand side of the figure (we will elaborate further on these strategies shortly). We analyzed the reordering of processes to ensure that it upholds valid execution sequences while maximizing processor utilization across each stage.

We employed three distinct parallelization strategies, delineated as follows.

1. Processes #0, #1, #10, and #19 are exclusively implemented in C++. Thus, the initial focus was on parallelizing these processes. Detailed explanations of how this was performed are furnished in subsequent sections.

2. Processes #2, #5, #8, and #17 are implemented in Fortran, and exhibit minimal execution times, prompting their parallelization through C++ OpenMP tasks.

3. Processes #9, #15, and #18, also implemented in Fortran, are characterized by the independent processing of distinct signal types, and plot generation. Consequently, these tasks were parallelized using C++ OpenMP tasks.

Using this strategy, we were able to parallelize 5 out of 11 stages, as depicted in Figure 2.6. The specifics of the parallelization for each stage are detailed below.

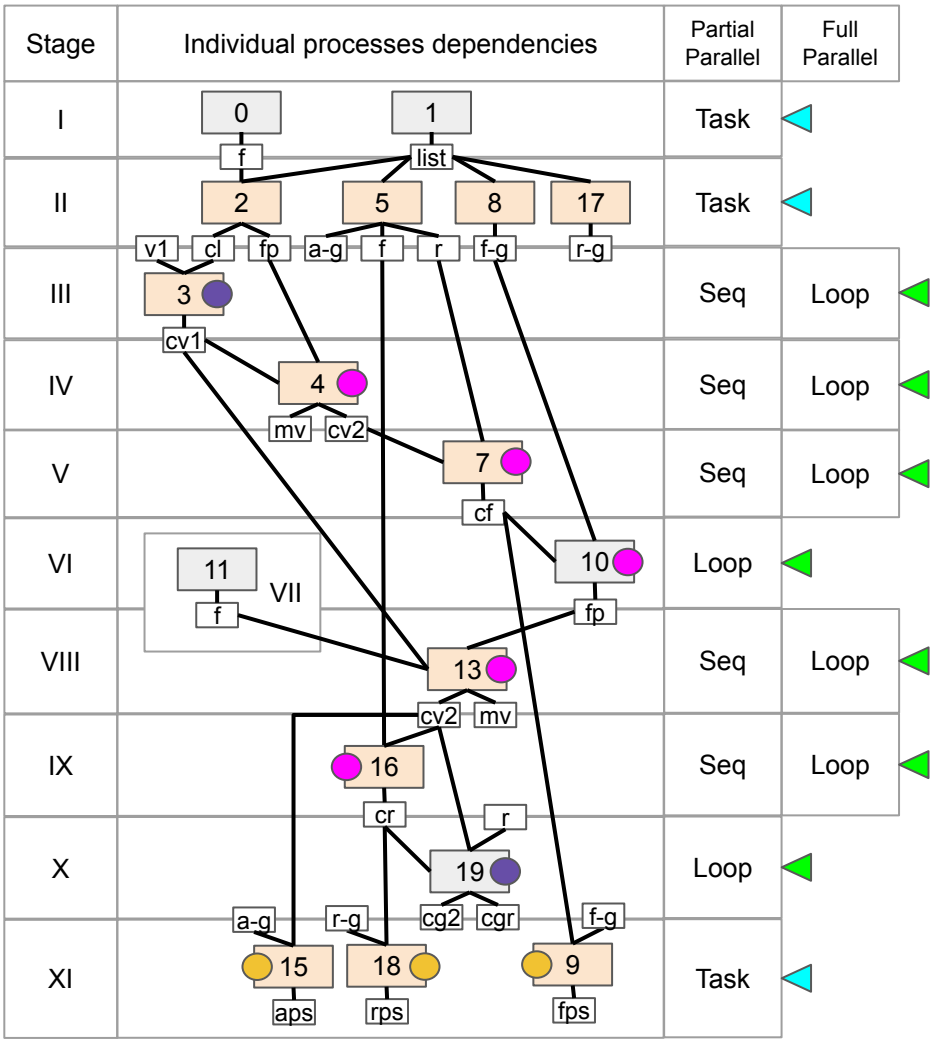



**Figure 2.6:** Processes reordering into 11 stages. Parallel approaches for each individual stage (rows) for the first and second parallel versions (right-most columns) are detailed, as well as input/output dependencies for each process.

### 2.5.1 Parallelizing Stages I and II

Since a limited number of equally lightweight processes are involved, we chose a task-parallelization approach. A parallel pragma is configured to utilize between 2 and 4 processors for task execution. The single pragma restricts task assignment to only the main processor. A task-wait pragma is necessary to synchronize the execution between both stages. In Stage I, processes operate inde-

pendently and do not require inputs. In Stage II, all processes create metadata files used in further stages based on the $[station][comp]$.v1 files list generated during Stage I.

```
#pragma omp parallel{
    #pragma omp single{
        #pragma omp task//Stage I processes:
        InitializeFlags();//P#0
        #pragma omp task
        GatherInputFiles();//P#1
        #pragma omp taskwait

        #pragma omp task//Stage II processes:
        InitializeFilterParams();              //P#2
        #pragma omp task
        InitMetadata(<acc-graph>,<fou>,<res>);//P#5
        #pragma omp task
        InitMetadata(<fourier-graph>);         //P#8
        #pragma omp task
        InitMetadata(<response-graph>);       //P#17
        #pragma omp taskwait
}   }
```

**Listing 2.1:** Stages I and II: C++ Task parallelism using 2 and 4 processors.

### 2.5.2 Parallelizing Stage VI

Process #10 identifies inflection points within the velocity Fourier spectrum for each of the three components. Concurrent analysis is conducted for each component. `CalculateInflection Point` within the parallel for-loop employs an early-termination strategy while searching for slope changes in data points for periods greater than one second. Although additional parallelization opportunities exist, they were not pursued, since execution time was already small.

```
void AnalyzeFourier(){
    //Read metadata (fourier-graph file)
    for(int i=0; i<N; i++){
        string files[3];//Input: <comp>.f files
        float fsl[3], fpl[3];
        //Analyze L, T, and V plots
        #pragma omp parallel for
        for(int j=0; j<3; j++)
            CalculateInflectionPoint(
                files[j], fsl[j], fpl[j]);
        //Output: FSL & FPL values (filter param)
}   }
```

**Listing 2.2:** Stage VI: C++ parallel for loop using 3 processors.

### 2.5.3 Parallelizing Stage X

Process #19 generates 18 *GEM* files for each input file. Specifically, six files are generated per *V2* and *R* file pair. Concurrent reading of each batch of files is facilitated in this stage, leveraging the maximum number of available processors, given the typically substantial quantity of *GEM* files produced. The function `SetDataApart` operates on each *V2* or *R* file within the parallel for-loop, generating three corresponding *GEM* files. This parallelized approach optimizes efficiency by distributing the workload across all available processors.

```
void GenerateGEMFiles(){
    //Read metadata (response file)
    string files[N*2];
    for(int i=0; i<N; i++){
        files[i*2] = //Input: <s>.v2 files
        files[i*2+1] = //Input: <s>.r files
    }
    #pragma omp parallel for
    for(int i=0; i<N*2; i++){
        bool isR = //flag (odd/even)
        SetDataApart(files[i], isR); //Output: <s><comp>GEM<2|R><A|V|D> files
}   }
```

**Listing 2.3:** Stage X: C++ parallel for loop using all available processors.

### 2.5.4 Parallelizing Stage XI

These plotting processes operate independently from one another and operate on different input data. They receive *V2*, *F*, and *R* files as inputs and generate corresponding plot files, namely [*station*].ps, [*station*]f.ps, and [*station*]r.ps.

```
#pragma omp parallel{
    #pragma omp single{
        #pragma omp task//Stage XI processes:
        PlotFourierSpectrum();//P#9
        #pragma omp task
        PlotAccelerograph();//P#15
        #pragma omp task
        PlotResponseSpectrum();//P#18
        #pragma omp taskwait
}   }
```

**Listing 2.4:** Stage XI: C++ Task parallelism using 3 processors.

Expanding on this, the *V2* files contain corrected signals, the *F* files store Fourier-transformed signals, and the *R* files comprise Response spectrum data. The resultant plots provide visual representations of the analyzed accelerographic data, aiding in interpreting and understanding the seismic event under investigation.

## 2.6 Fully Parallelized Implementation

The parallelization of the remaining processes requires either Fortran OpenMP pragmas or executing binary files concurrently within temporary folders. This implies the transfer of all input and output data to and from these folders. We employed two distinct approaches:

- The parallelization approach for Processes #3 and #16 is via Fortran OpenMP do-loops, enabling efficient concurrency within the existing Fortran codebase.
- Concurrent execution of processes #4, #7, and #13 inside temporary folders with required data transfer is achieved through C++ OpenMP for-loops.

All processes except #11 are parallelized (Fig. 2.7), due to its execution time being less than two milliseconds on average. We describe specific approaches used in each stage.

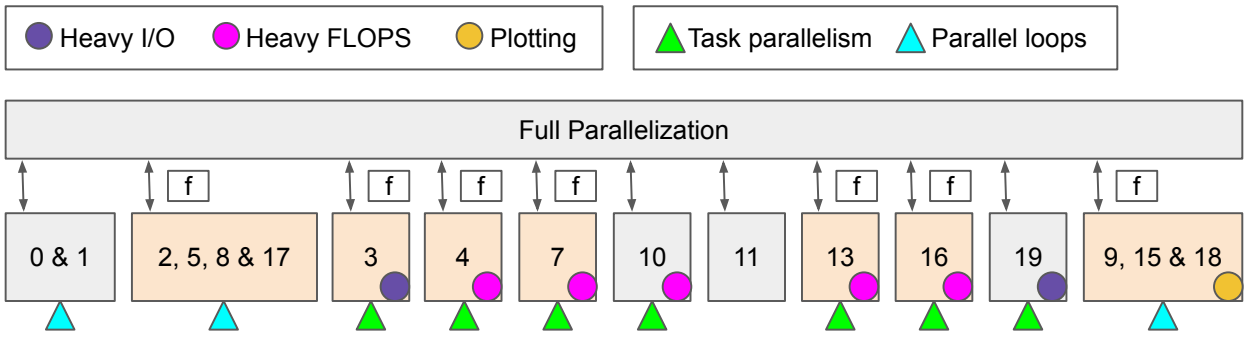



**Figure 2.7:** Full parallelization: all stages are executed in parallel.

### 2.6.1 Parallelizing Stage III

Process #3 concurrently collects uncorrected acceleration data from all $[station]$.v1 files and generates an individual file $[station][comp]$.v1 for each component.

This task is accomplished using Fortran OpenMP pragmas, including `omp parallel` and `omp do`, which leverage all available processors for enhanced parallel execution efficiency.

```
c   Read data (<station><comp>.v1 list file)
C$OMP PARALLEL
C$OMP DO PRIVATE(<variables>)
do i=1,N
c   Read acceleration data
open(unit=i*4,file=..,status='old') -> Input:<s>.v1
c   Write acceleration data per component
open(unit=i*4+1,file=...) -> Output: <s>l.v1
open(unit=i*4+2,file=...) -> Output: <s>t.v1
open(unit=i*4+3,file=...) -> Output: <s>v.v1
end do
C$OMP END DO
C$OMP END PARALLEL
```

**Listing 2.5:** Stage III: Fortran parallel do loop using all available processors.

### 2.6.2 Parallelizing Stage IX

Process #16 starts by extracting metadata from the response file, that is, the list of individual component $[station][comp]$.v2 file names. Then, each file is read in parallel. The elastic response spectra for acceleration, velocity and displacement are calculated for each component.

It is important to note that this process has a sequential complexity of $O(9000 * N * D^2)$ where $N$ is the number of input *V1* files, and $D$ is the average number of individual data points contained in each *V1* file. Lastly, output data is stored in individual $[station][comp]$.r files for each component. The response spectrum data is used in processes #18 and #19, which generate plots and create individual *GEM* files, respectively.

This parallelization is particularly significant for this process, because it not only takes the longest to execute, but also achieves the highest speedup, as shown in Figure 2.8 of the experimental results. We use all available processors to maximize efficiency in this optimization effort.

```
c   Read metadata (response file)
C$OMP PARALLEL
C$OMP DO PRIVATE(<variables>)
do i=1,<3N> -> Each component in parallel
c   Read acceleration data -> Input: <s><comp>.v2
open(unit=2*i,file=...,status='old')
c   Calculate Response Spectrum
c   Write output data
open(unit=2*i+1,file=...) -> Output: <s><comp>.r
end do
C$OMP END DO
C$OMP END PARALLEL
```

**Listing 2.6:** Stage IX: Fortran parallel do loop using all available processors.

### 2.6.3 Parallelizing Stage V

Process #7 is responsible for computing the Fourier spectra for acceleration, velocity, and displacement of each component stored in all $[station][comp]$.v2 files, with the results saved in $[station][comp]$.f files. The parallelization strategy employed for this process mirrors that of Stages IV and VIII, as detailed in the preceding section. All available processors are utilized to optimize computational efficiency.

```
void ParallelizeFourier(){
  //Read data (fourier file)
  string files[N*3];
  for (int i = 0; i < N; i++)
      files[3*i] = //Input: Folder name
      files[3*i+1] = //Input: <s><comp>.v2 files
      files[3*i+2] = //Input: <s><comp>.f files

  #pragma omp parallel for
  for (int i = 0; i < N; i++)
      //Create temp 3*i folder & fourier file
  for (int i = 0; i < N; i++)//Seq. to avoid races
      //Move EXE to 3*i folder
  #pragma omp parallel for
  for (int i = 0; i < N; i++)
      //Input: Move 3*i+1 <s><comp>.v2 file
      //Apply Fourier Transform on 3*i folder
      //Output: Move 3*i+2 <s><comp>.f file
      //Delete remaining temp files
}
```

**Listing 2.7:** C++ parallel for loops using all available processors.

### 2.6.4 Parallelizing Stages IV and VIII

Processes #4 and #13 share the same functionality. Specifically, they read the filter parameters file to extract the values of FPL and FSL, and take all the individual $[station][comp]$.v1 files as input. Subsequently, a Hamming band-pass filter is applied to each signal, resulting in a corrected version of the acceleration values. These corrected values are then stored in individual $[station][comp]$.v2 files.

```
void ParallelizeCorrection(){
  //Read data (filter params file)
  string files[N*10];
  for (int i = 0; i < N; i++)
      files[10*i] = //Input: <s><comp>.v1 files
      for (int j = 0; j < 3; j++)
          for (int k = 1; k <= 3; k++)
              getline(i_data, files[10*i+3*j+k]);
  #pragma omp parallel for
  for (int i = 0; i < N; i++)
      //Create temp 10*i folder and params file
      for (int j = 0; j < 3; j++)
          for (int k = 1; k <= 3; k++)
              //Move 10*i+3*j+k <s><comp>.v1 file
  for (int i = 0; i < N; i++)//Seq. to avoid races
      //Move EXE to 10*i folder
  #pragma omp parallel for
  for (int i = 0; i < N; i++)
      for (int j = 0; j < 3; j++)
          //Input: Move 10*i+3*j+2 <s>.v1 file
      //Apply filters to signals on 10*i folder
      for (int j = 0; j < 3; j++)
          //Output: Move 10*i+3*j+3 <s>.v2 file
  //Output: (max values file)
  #pragma omp parallel for
  for (int i = 0; i < N; i++)
      //Delete remaining temp files
}
```

**Listing 2.8:** C++ parallel for loops using all available processors.

However, the parallelization strategy for these processes differs significantly from the approaches used in the other stages. Due to the impracticality of modifying the original Fortran programs, multiple instances are executed concurrently within separate folders. The primary task involves creating these folders with all the necessary files, and subsequently copying the results back. This approach maximizes processor utilization by harnessing all available processors.

## 2.7 Experiments and Results

### 2.7.1 Experimental Setup

Our experimental dataset comprises 71 unprocessed accelerograph files from 6 seismic events that occurred in El Salvador over the past decade. These events represent a diversity of stations affected, and have varying numbers of data points within each raw file, ranging from 7,300 to 35,000. The original sequential implementation consisted of 2,297 lines of Fortran code. Our fully-parallelized version was implemented with 653 lines of C++ code and an additional 337 lines of Fortran code. The experimental platform used for the experiments was a 12th Gen Intel Core i5-12450H, 2.00 GHz, 8 Cores, 12 Logical Processors, 16 GB of RAM, 640 KB of L1 cache, 6.2 MB of L2 cache, and 12 MB of L3 cache.

### 2.7.2 Results per Stage

For this experiment, we assessed the parallel performance of each individual stage using data from the seismic event with the most data in the experimental dataset, i.e., $384,000$ data points distributed across $19$ *V1* files. Figure 2.8 illustrates that stage IX exhibits the longest execution time among all stages, accounting for 57.2% of the original sequential implementation. However, Stage IX also achieves the highest speedup, reaching 5.14x. Other parallel stages successfully reduce the sequential execution time by more than half, achieving speedups of 2.2x, 2.0x, 2.6x, and 2.1x for stages I-II, IV, VI, and XI, respectively. Furthermore, the remaining parallel stages achieve speedups of 1.8x, 1.7x, 1.9x, and 1.5x for stages III, V, VIII, and X, respectively. Overall, for this event, the speedup stands at 2.88x, as indicated in the bottom row of Table 2.1.

### 2.7.3 Results per Event

All available uncorrected accelerographs (*V1* files) for each seismic event in the experimental dataset were processed using the implementations explained in sections III to VI: *Sequential Original*, contains 20 sequential processes; *Sequential Optimized*, contains 17 sequential processes; *Partially Parallelized*, utilizes 5 parallel stages; and *Fully Parallelized*, utilizes 10 parallel stages.

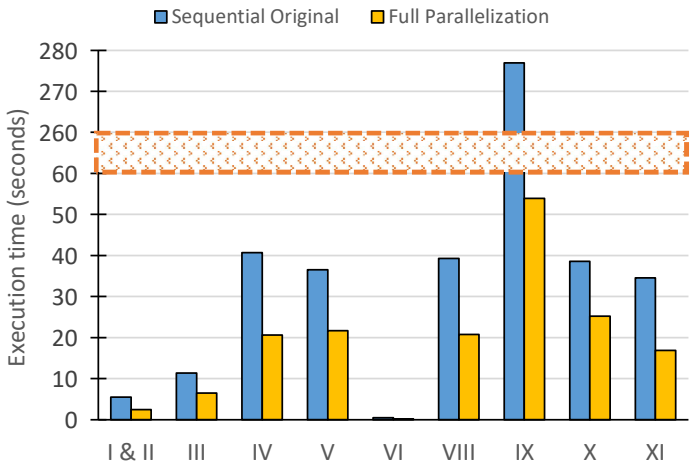



**Figure 2.8:** Speedup per individual Stage (19 Files, 384k Data points)

Figure 2.9 shows a visual representation of each implementation's execution times, and full data is shown in Table 2.1. Execution time is linearly proportional to the total amount of data points. While each implementation offers a performance improvement over its predecessor, the fully-parallelized version demonstrates the most efficiency.

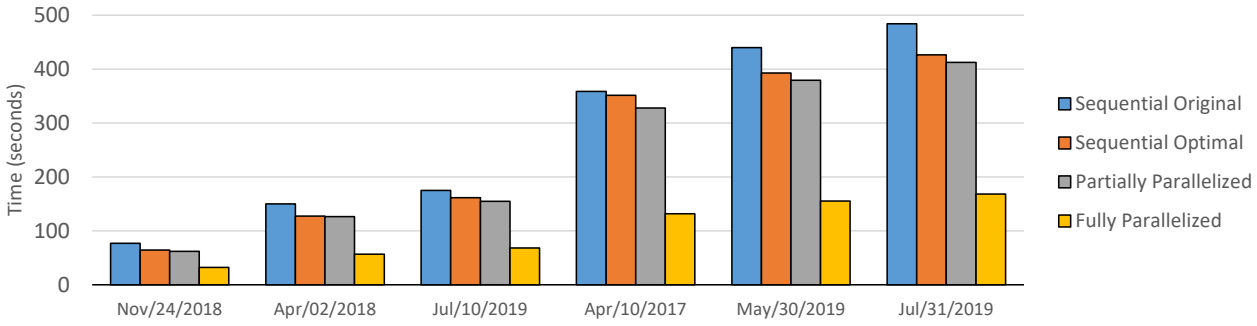



**Figure 2.9:** Speedup per Event: exec. time is proportional to number of data points in each seismic event.

**Table 2.1:** Experimental Results. *Execution times are measured in seconds.

| Event | V1 Files | Data Points | Seq. Ori.* | Seq. Opt.* | Part. Par.* | Full Par.* | Speed Up |
|---|---|---|---|---|---|---|---|
| Nov'18 | 5 | 56K | 76.6 | 64.1 | 61.9 | 32.1 | 2.39x |
| Apr'18 | 5 | 115K | 149.6 | 127.1 | 126.4 | 56.5 | 2.65x |
| Jul'19 | 9 | 145K | 174.9 | 161.3 | 154.8 | 68.1 | 2.57x |
| Apr'17 | 15 | 309K | 358.6 | 351.2 | 327.9 | 131.5 | 2.73x |
| May'19 | 18 | 361K | 439.5 | 392.6 | 378.9 | 155.3 | 2.83x |
| Jul'19 | 19 | 384K | 483.7 | 426.0 | 412.2 | 168.1 | 2.88x |

We observe that the speedup increases proportionally with the number of total input data points, as illustrated in Figure 2.10. The overall speedup ranges from 2.4x to 2.9x and appears to follow a quasi-logarithmic trend.

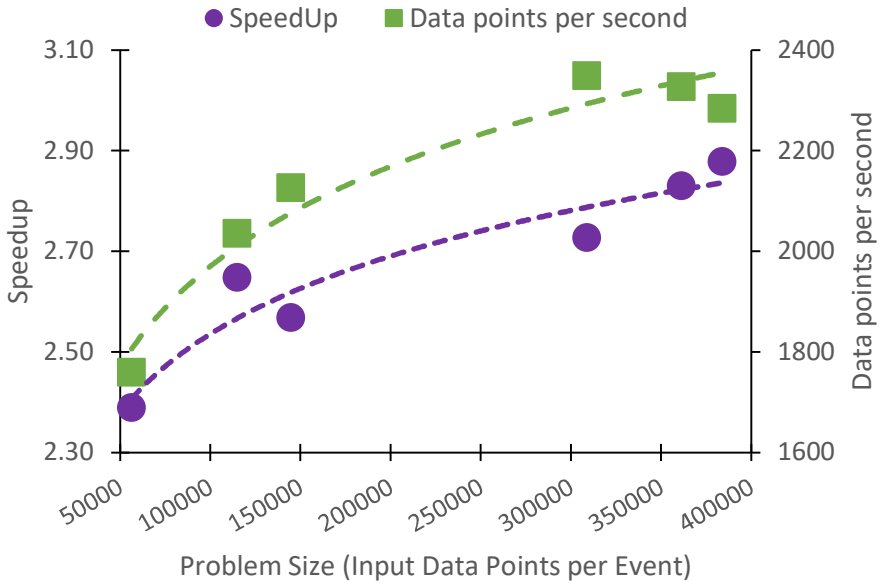



**Figure 2.10:** Overall speedup in parallel accelerographic records processing: problem size vs. speedup (purple) and vs. data points per second (green).

The fully-parallelized implementation was able to process between 1,700 to 2,300 data points per second, a significant improvement over the original sequential version, which processed 800 data points per second on average. This improvement also follows a quasi-logarithmic trend on the problem size, as shown in Figure 2.10.

## 2.8 Discussion & Future Work

Although the number of seismic events contained in our experimental dataset is relatively small, it offers a comprehensive range of real-world seismic data, generated in different geographic locations with a variety of equipment types and sampling rates. Although we used a relatively low-end machine for the experiments, and performance may be further improved on a higher-performance machine, our experimental platform is reflective of the types of machines used in practice in the Salvadoran Observatory of Natural Threats. It is worth noting that the engineering work invested in parallelizing each stage does not always directly translate to performance gains. Despite sim-

ilar levels of effort, Stage IX provides the most significant improvements. Additionally, different stages leverage available processors differently, with some using only a fraction while others fully utilize all available resources. Our findings indicate potential for leveraging tools like OpenMP to enhance Accelerographic Records Processing, though communication overhead remains a concern. We also observed similarities in array management techniques across stages IV, VIII, and V, resembling principles seen in MPI or CUDA programming.

While we believe our approach to be an important step toward the high-level goal of scaling up strong-motion records processing, especially in regard to large-scale real-world datasets, there are engineering and research challenges we plan to address in future work. Enhancing strong-motion record processing could involve automatic translation of diverse legacy code into a single programming language, such as C++, and using search-based techniques [11–13] to assist in parallelization of the code, which could streamline development and improve overall efficiency. With adequate time and resources, similar enhancements could extend to other seismic-related applications, as well as processes related to other natural threats, like vulcanology and landslides.

Additionally, there is room for further optimization through the exploration of advanced parallel processing techniques like tiling, wavefront scheduling, and the polyhedral model. Furthermore, scaling our approach to larger experimental accelerographic datasets presents an exciting opportunity to assess its performance and scalability in real-world scenarios.

## 2.9 Related Work

### 2.9.1 Strong-Motion Records Databases

Several initiatives worldwide have been dedicated to collecting and disseminating strong-motion data to advance seismic research and hazard assessment. For instance, the ITACA project focused on gathering, standardizing, and sharing strong motion data acquired in Italy since 1972. By 2010, this database encompassed 7,038 waveforms from analog and digital instruments recorded during 1,019 earthquakes with magnitudes reaching 6.9 [14]. Similarly, the Salvadoran Accelerographic Repository stores 6,787 strong motion records from 1,615 seismic events between the years 1966 and 2019. It scales up quickly; the most recent report shows 241 seismic events recorded just during December 2023 [15]. Another example is the Indian Strong Motion Instrumentation Net-

work, with around 220 accelerograph stations, which provides data from approximately 300 strong ground motion records from 130 earthquakes [5]. Furthermore, a comprehensive database has been established to study induced earthquakes in the Groningen gas field, Netherlands. This repository houses over 8,500 processed ground motion recordings from 87 earthquakes, serving as a resource for refining seismic hazard and risk models and conducting research in site response and ground motion characteristics [16].

### 2.9.2 Earthquake Engineering and Urban Planning

Despite efforts to enhance safety through urban planning, indiscriminate development and structural failures persist, leading to significant loss of life and economic damage [17]. To address these challenges, there is a need to prioritize the development of resilient buildings and emergency response efforts by mapping vulnerable urban areas and populations. While progress has been made in evaluating individual facilities and distributed systems, greater integration and collaboration are necessary to improve overall seismic resilience [18]. In El Salvador, the original sequential code discussed in this paper was a component of a large project that united academia, industry, and governmental agencies with the goal of updating building design codes. Various software tools are utilized in earthquake engineering and seismic risk assessment, enabling different aspects of the analysis process. High-performance computing clusters enable parallel job configurations for handling buildings independently [18]. Neural networks like the multi-layer perceptron assess urban block vulnerability to earthquakes [17]. Additionally, software tools such as Obspy [19, 20] and TSPP [21] are employed for ground-motion processing, aiding in structural design and seismic evaluation [4].

### 2.9.3 Seismic and Volcanic Observatories

Software plays a crucial role in supporting natural-threat observatories by processing strong-motion records, like the ones from El Salvador used in our experiments. A parallel approach, similar to ours, using Python and MPI, enables efficient processing of strong-motion files, albeit from volcanoes, demonstrating scalability with the number of events and processor cores [22]. In contrast, sequential software like Earthworm (developed by the USGS) and packages from the Alaska Volcano Observatory provide robust tools for seismic data analysis and real-time monitor-

ing [7, 23]. Additionally, the waveform suite from the University of Alaska Geophysical Institute offers MATLAB code for waveform data manipulation, ensuring data integrity and program stability [24]. These software tools are vital for monitoring seismic and volcanic activity and conducting research in these fields.

### 2.9.4 Early Warning Systems

Software supporting early warning systems for earthquakes and tsunamis plays a pivotal role in mitigating the impact of these natural disasters. The German Indonesian Tsunami Early Warning System (GITEWS) Project has developed the SeisComP3 software package, which offers reliable and fast earthquake location and magnitude estimation capabilities [1]. SeisComP3 is also used in El Salvador as a real-time support tool that allows seismic technicians to visualize and handle customized parameters for magnitude calculations. Furthermore, machine learning approaches, such as Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks, are utilized in Early Earthquake Warning (EEW) systems to issue alerts promptly [25]. Early warning systems require rapid data processing and transmission to provide timely alerts, with mere seconds dedicated to sensor data acquisition, processing, magnitude estimation, and decision transmission to the warning server [2, 26, 27]. Overall, developing and deploying advanced software solutions are essential for enhancing the effectiveness of early warning systems, thereby reducing the impact of earthquakes and tsunamis on vulnerable communities.

### 2.9.5 General purpose seismology research

Parallel programming plays a crucial role in advancing seismology research by enabling the efficient processing of strong-motion datasets. Computing Grids provide a platform for sharing seismic data from diverse sources, facilitating seismic waveform analysis, and uncovering observed regions' geological features. The authors of [28] propose a framework for massively parallel wavelet data processing of seismic waveforms. Software tools like *Seismic Analysis Code* offer capabilities for processing multiple signals concurrently, enhancing research productivity and enabling detailed seismic event analysis [29, 30]. A server-side tool [31] allows parallel processing of accelerometric waveforms through a combination of Python for signal processing, Fortran for parallel acceleration and displacement response spectrum calculation, and PHP for dynamic plot

generation. Moreover, similar to our approach's architecture, efforts to parallelize source code using High-Performance Fortran and OpenMP demonstrate a faster and more accessible approach to simulating seismic radiation interactions with near-surface geological structures [8].

Nowadays, a considerable portion of scientific software operates sequentially. For instance, the SEISAN seismic analysis system provides a comprehensive suite of programs written primarily in Fortran, supplemented by some C code, facilitating earthquake analysis from both analog and digital data sources [32]. Another example is the open-source software PASCAL Quick Look eXtended (PQLX), which allows for detailed analysis of seismographic data records to study ambient noise and detect earthquake events reliably [33]. Additionally, MATLAB software like EQK_SRC_PARA enables estimation of earthquake source spectrum spectral parameters, which are essential for analyzing seismic events and developing scaling laws for specific study regions [34]. These programming tools offer valuable support for seismologists in analyzing seismic data, understanding earthquake characteristics, and improving earthquake detection systems.

## 2.10 Conclusion

Our paper introduces a fully-parallelized approach for strong-motion record processing, marking a significant advancement over the original sequential version. Leveraging parallel loops and task parallelization, our method effectively addresses the challenge of efficiently processing accelerographic data, providing scalability and speedup roughly proportional to problem size. This demonstrates the potential for parallel programming techniques to enhance scientific software, playing a pivotal role in advancing seismology research. By enabling efficient data processing, analysis, and simulation of seismic events, such software contributes to a deeper understanding of seismic phenomena and aids in mitigating their impact on society.

## Acknowledgments

We thank Sanjay Rajopadhye and William Scarbro for helpful feedback on an earlier version of this work. We are grateful to the anonymous reviewers for their detailed feedback on this paper. The Colorado State University co-authors were partially supported by the US National Science Foundation under grant No. 2318970.

# Chapter 3

# Heterogeneity-Aware Software Performance Characterization via Graph Machine Learning

## 3.1 Introduction

Software is becoming increasingly complex, longer, and more challenging to analyze. As discussed in the previous chapter, manual optimization can sometimes suffice for time-critical systems since some software is repeatedly executed invariably in the same environment.

However, as heterogeneous hardware becomes pervasive in research and industry, automatic optimization becomes increasingly difficult. There is a growing need for not only fast but also heterogeneity-aware methods for performance estimation to guide the optimization of these time-critical systems. This chapter will explore how to achieve this through graph machine learning.

### 3.1.1 Motivation and technical challenges

As detailed in the following sections, there are three main challenges in using graph machine learning to estimate the performance of software in a heterogeneity-aware manner. First, minor changes in source code can result in significant performance variations, yet traditional source code-to-graph representation models like ASTs and SSAs fail to capture these subtle differences.

Second, graph machine learning requires representation learning algorithms to convert graphs into embedding vectors. While state-of-the-art graph embedding methods are effective, they are computationally expensive and resource-intensive to train.

Third, machine learning needs large amounts of training data, but there is a lack of publicly available datasets suitable for graph machine learning that are designed with heterogeneity in mind. The more unique architectures are included and used to perform simulations and gather performance metrics, the better and more robust machine learning models can be trained, thus allowing them to generalize across new hardware architectures.

### Motivation example

The following code demonstrates a doubly nested for loop, with a minimal change in the order of loop execution. This slight modification can significantly impact the performance of a program handling large datasets in memory due to cache locality effects. Unfortunately, the most commonly used graph representations for source code fail to produce distinct graphs for these two programs. Both representations have the same number of nodes, and the edges connect identical nodes.

```
1 int main() {
2     int matrix[2][3] =
3             {{1,2,3},{4,5,6}};
4     int i, j;
5     for (i = 0; i < 2; i++)      // I
6         for (j = 0; j < 3; j++) // J
7             matrix[i][j]++;
8     return 0;
9 }
```

**Listing 3.1:** Motivation example IJ.

```
1 int main() {
2     int matrix[2][3] =
3             {{1,2,3},{4,5,6}};
4     int i, j;
5     for (j = 0; j < 3; j++)      // J
6         for (i = 0; i < 2; i++) // I
7             matrix[i][j]++;
8     return 0;
9 }
```

**Listing 3.2:** Motivation example JI.

The only difference appears in the Static Single Assignment (SSA) form, where the content of a node varies slightly. This variation is only detectable by specific graph embedding approaches considering node content. However, most graph embedding algorithms focus on structural aspects, making them incapable of distinguishing between these two performance-differentiated programs.

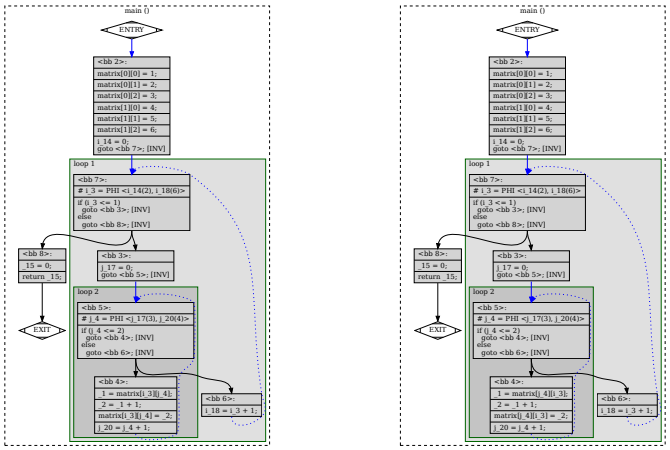



**Figure 3.1:** SSA graph representation obtained with Motivation example IJ (left) and JI (right).

## 3.2 Background

### 3.2.1 Graph Machine Learning

Graph Machine Learning (Graph ML) aims to make predictions or discover patterns using graph-structured data as input. Some fields where it is used nowadays are: computer vision, NLP, cybersecurity, bioinformatics, social media analysis, recommendation systems, smart cities, digital humanities, semiconductor manufacturing, weather, air quality, and seismology.

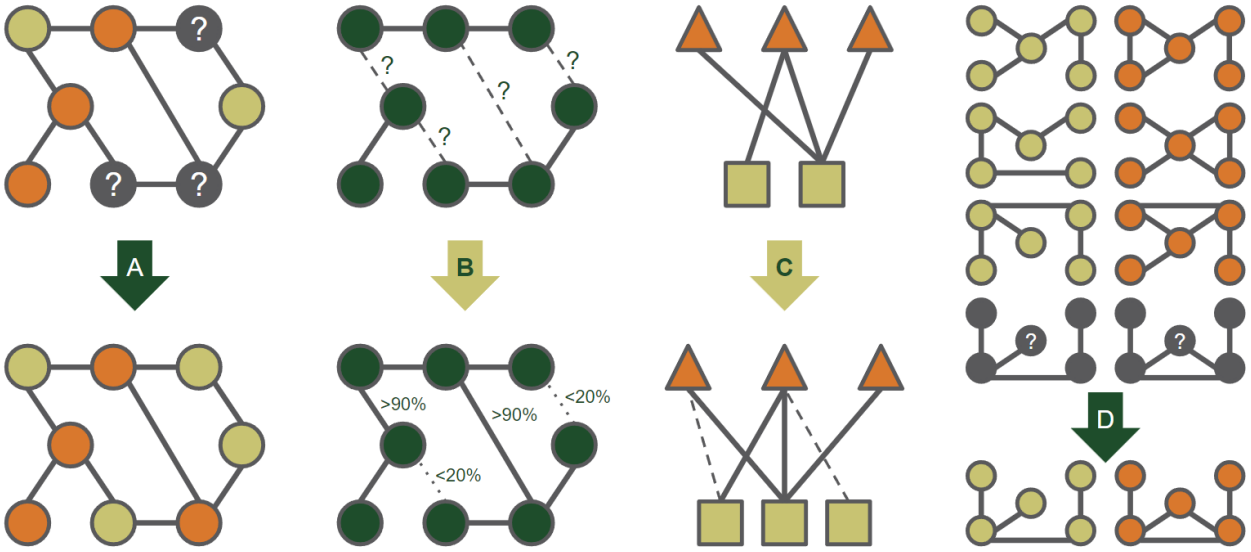



**Figure 3.2:** Overview of Graph Machine Learning common tasks: (A) node classification, (B) link estimation, (C) recommendation systems, and (D) graph classification.

### 3.2.2 Overview of Graph Representation Learning

The central problem in GraphML is finding a way to incorporate info about graph structures into the ML models. The challenge from an ML perspective is that there is no straightforward way to encode this high-dimensional, non-Euclidean information contained in a graph structure into a feature vector [35]. The complexity of graph data entails significant challenges to existing ML algorithms. For example, the common premise that instances are independent of each other no longer holds for graph data because each instance, aka nodes, is related to others by links [36].

**Figure 3.3:** Machine learning with graphs: Graph Representation Learning.

To use graphs in downstream machine learning and data mining applications, graphs and their entities, such as nodes and edges, need to be represented using numerical features [37]. A way to represent a graph is its adjacency matrix. However, its size is often very high in real-world graphs.

Mapping graph entities to low-dimensional vectors while preserving intrinsic graph properties like graph structure and entity relationships is the goal of graph representation learning, also known as graph embedding. GRL aims to obtain vector representations of graph entities (e.g., nodes, edges, subgraphs, etc.) for its use in downstream tasks [38]. Obtaining an accurate representation from raw graph data is a challenging task, as choosing the proper graph property to embed is an issue of concern if a graph has many properties [39].

DEFINITION 1. *Let $G = (V, E)$ be a graph where $V$ and $E$ are the set of nodes and the set of edges of the graph, respectively. Node embedding is a mapping function $f : v_i \rightarrow \mathbb{R}^d$ that encodes each graph's node $v_i$ into a low dimensional vector of dimension $d$ such that $d \ll |V|$ and the similarities between nodes in the graph are preserved in the embedding space.*

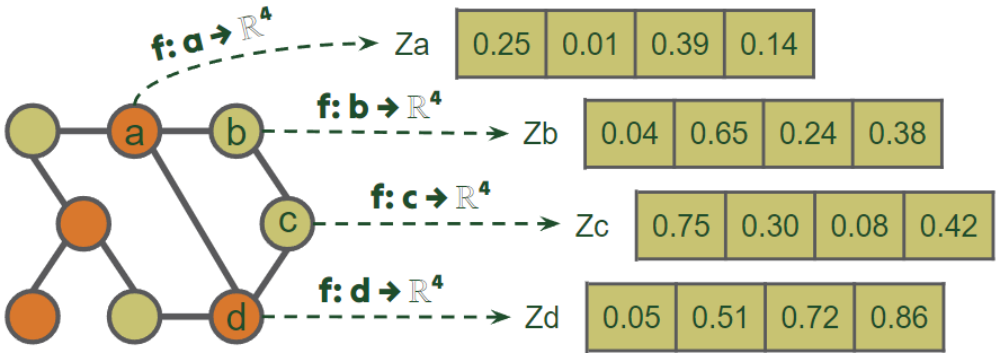



**Figure 3.4:** Node embeddings. Adapted from [37]

While edge, subgraph, and graph embeddings can be learned directly from graphs, most commonly, they are derived from node embeddings [37]. This downsampling strategy is called a pooling operation. It is mainly used to generate graph-level representations based on node vectors [36].

DEFINITION 2. *Let $S = \{G_1, G_2, \ldots G_n\}$ be a set of graphs. Graph embedding is a mapping function $f : G_i \rightarrow \mathbb{R}^d$ that converts each graph of $S$ into a low dimensional vector of dimension $d$ such that $d \ll |S|$ and the similarities between graphs in the set are preserved in the embedding space. Note that graphs do not have to be of the same size (amount of nodes) between them.*

Whole graph embedding is usually done for small graphs such as proteins and molecules. These smaller graphs are represented as one vector, and two similar graphs are embedded to be closer. Whole-graph embedding facilitates graph classification tasks by providing a straightforward and efficient solution for computing graph similarities [39].

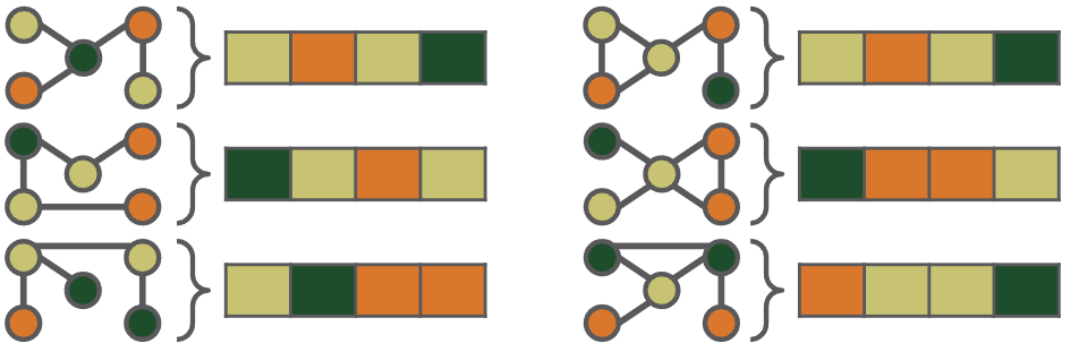


**Figure 3.5:** Graph embeddings.

Visualizing embeddings is useful to understand how well the embedding methods learn from the graph structure and node attributes. To be considered a good representation, vectors for similar nodes should have shorter distances from each other; one way of calculating this is through cosine similarity. It is common to use t-SNE [40], a widely-used dimension reduction algorithm, to transform high-dimensional vectors into a two-dimensional space for visualization.

As the number of embedding dimensions decreases, less information about the input graph is preserved; thus, downstream task performance will naturally drop. However, the decrease in embedding size affects some methods more than others [39].

### 3.2.3 Overview of Graph embedding algorithms

A graph and its elements can be represented for a particular application using carefully hand-crafted features. For example, in anomaly detection applications, the nodes with the densest neighborhood are more likely to be anomalous. Therefore, if we include the in-degree and out-degree in the node representation, we can detect the anomalous nodes with high accuracy [37].

Extending on this idea, Graph Embedding typically refers to techniques that map nodes, links, or entire graphs to a lower-dimensional vector space in an unsupervised manner. These embeddings are used as features for downstream tasks in machine learning models.

#### Bag of nodes

The simplest approach to defining a graph-level feature is to aggregate node-level statistics. For example, one can compute histograms or other summary statistics based on the degrees, centralities, and clustering coefficients of the nodes in the graph. This aggregated information can then be used as a graph-level representation. The downside to this approach is that it is entirely based upon local node-level information and can miss important global properties in the graph [41].

#### Random walks

These methods generate random walks for each node in the graph to capture the structure of the graph and output similar node embedding vectors for nodes that occur in the same random walks.

DEFINITION 3. *In a graph $G = (V, E)$, a random walk is a sequence of nodes $v_0, v_1, \ldots, v_n$ that starts from node $v_0$. $(v_1, v_{i+1}) \in E$ and $k + 1$ is the length of the walk. The next node in the sequence is selected based on a probabilistic distribution.*

DeepWalk [42] and Node2vec [43] are based on the Word2vec embedding method [44] in NLP. Word2vec is based on the observation that words that often co-occur in the same sentence have a similar meaning. Node2vec and DeepWalk extend this assumption for graphs by considering that nodes that co-occur in random walks are similar. The difference is that DeepWalk selects the next node in the random walk uniformly from the neighbor nodes of the previous node, while Node2vec biases the walk based on two parameters, making it random, BFS, or DFS-like.

**Graph kernels**

Graph kernels aim to compare graphs or their substructures (e.g., nodes, subgraphs, and edges) by measuring their similarity [45]. This section focuses on two methods: graphlet and WL.

**Graphlet kernels** measure the similarity between graphs by counting subgraphs with a limited size k [46] [47]. Formally, a graphlet kernel involves enumerating all possible graph structures of a particular size and counting how many times they occur in the full graph [41]. The challenge with this approach is that counting these graphlets is a combinatorially difficult problem, though numerous approximations have been proposed [48].

**The Weisfeiler-Lehman kernel** uses a strategy of iterative neighborhood aggregation. This approach aims to extract node-level features that contain more information than their local ego graph and then aggregate these richer features into a graph-level representation [41] [49] [38].

This is considered to be a traditional strategy to test the homomorphism of two graphs using color refinements. Several models improved the idea from the WL isomorphism test [50] [51]. This concept inspired various GNN models later, which aim to be expressive as powerful as the WL test to distinguish different graph structures [52].

**Algorithm 1** 1-WL (color refinement)

**Require:** $G = (V, E, X_V)$
1: $c_v^0 \leftarrow \text{hash}(X_v) \forall v \in V$
2: **while** $(c_v^l) v \in V \neq (c_v^{l-1}) v \in V$ **do**
3: $\quad c_v^l \leftarrow \text{hash}(c_v^{l-1}, \{\{c_w^{l-1} : w \in N_G(v)\}\}) \forall v \in V$
4: **return** $\{\{c_v^l : v \in V\}\}$

Graph kernels are effective models with several advantages and disadvantages. For example, naive implementations require massive resources since these methods are NP-hard; heuristics and tricks are required for them to be scalable. Also, the above methods only focus on homogeneous graphs in which nodes do not have side information. In the real world, graph nodes could contain labels and attributes and change over time, making it challenging to learn node embeddings [38].

### Proximity Reconstruction

Large-scale information network embeddings (LINE) [53], is a shallow embedding approach. It generates node embeddings that preserve the first and second-order proximities in the graph, assuming that nodes with many common connections are similar. LINE trains two models that minimize $L_1$ and $L_2$ separately, and then concatenates them to obtain the final embedding of a node [37]. However, the global structure of graphs cannot be captured this way. A few variations try to capture the higher-order proximity of nodes, but computational complexity remains a problem.

### Deep Neural Network-Based

In recent years, large-scale graphs have challenged the ability of numerous graph embedding models. Traditional models, such as shallow neural networks, cannot efficiently capture complex graph structures. Based on the model architecture, models can fit into four main groups: graph autoencoders, recurrent GNNs, convolutional GNNs, and graph transformer models.

Structural Deep Network Embedding (SDNE) [54] is based on an autoencoder that reconstructs a graph's adjacency matrix and captures nodes' first and second-order proximities. Unlike shallow embedding, SDNE directly incorporates graph structure into the encoder using DNNs.

### Spectral Graph Embedding

Spectral methods operate in the spectral domain and involve leveraging the eigenvalues and eigenvectors of the graph Laplacian. These methods often require computing the graph Fourier transform. When computing power is insufficient for implementing convolution operators directly on the graph domain, several studies focus on transforming graph data to the spectral domain and applying filtering operators to reduce computational time [55] [56] [57]. NOn-Backtracking Embedding (NOBE) and its approximation (NOBEGA) are commonly used approaches [58].

### Sub2Vec: Feature Learning for Subgraphs

Related to graph classification, [59] suggests learning the embedding of each graph by treating them as a subgraph of a union of all the graphs. Then, leverage off-the-shelf classifiers.

### 3.2.4 Graph representation of source code

As discussed earlier in this chapter, preprocessing is essential before applying machine learning techniques to graphs. Raw inputs must be transformed into graph structures. This study's primary input data is source code written in C and C++. The scientific literature offers several well-known graph representations for source code. This section will describe the most commonly used ones.

#### Abstract Syntax Trees (AST)

A tree representation of the abstract syntactic structure of source code. Each node represents a construct, such as expressions, statements, or declarations. They clearly and concisely represent the code's structure, making it easier for compilers to perform syntax checking, optimization, and code generation. Additionally, ASTs are useful for refactoring tools and code analysis tools that need to understand the high-level structure of the code. However, a disadvantage is that ASTs can become complex and difficult to manage for large codebases, especially when handling programming languages with intricate syntax rules. Moreover, they abstract away some syntactical details, which might be necessary for certain types of analysis, leading to the potential loss of information.

#### Control Flow Graphs (CFG)

A graph which illustrates the order in which individual instructions, statements, or blocks of code are executed within a program. Each node in a CFG represents a basic block, which is a sequence of consecutive statements or instructions without any branches, except at the end. The edges between nodes indicate the potential flow of control from one block to another, capturing the various paths a program might take during its execution.

The advantages of CFGs include their utility in various compiler optimizations and program analysis tasks. They help detect unreachable code, identify loops and conditions, and optimize the execution path. CFGs are crucial for static analysis tools that seek to identify potential errors and vulnerabilities in code by examining all possible execution paths. However, the disadvantages lie in their potential complexity for large programs, where the number of possible paths can become overwhelmingly large. This complexity can make CFGs challenging to construct, visualize, and

analyze. Additionally, CFGs primarily focus on the flow of control and do not capture the data dependencies between variables, which might be necessary for more comprehensive analyses.

**Static Single Assignment (SSA)**

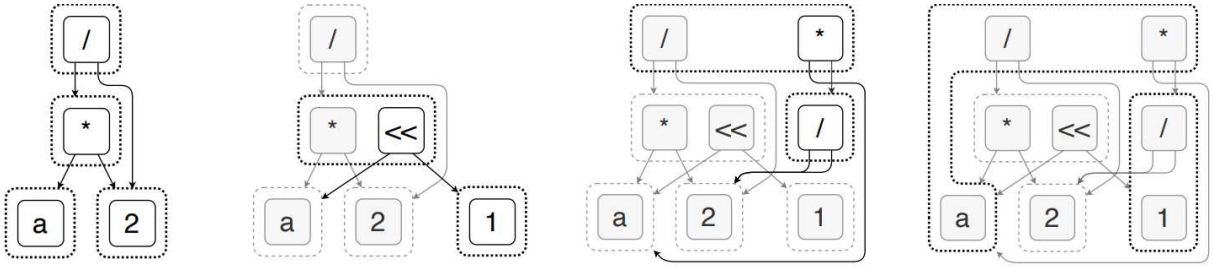
SSA is a property of an intermediate representation (IR) in which each variable is assigned exactly once, and every variable is defined before it is used. This is achieved by renaming variables so that each assignment to a variable has a unique name. In SSA form, a control flow graph (CFG) is annotated with $\phi$ (phi) functions, merging different variable values from different control flow paths. What distinguishes SSA from CFG is that while CFG focuses on the program's control flow, SSA emphasizes the data flow by ensuring each variable assignment is unique and explicitly defined. This unique assignment property simplifies various compiler optimizations, such as constant propagation, dead code elimination, and common subexpression elimination, as it makes data dependencies more explicit and easier to analyze. However, the main disadvantage is that converting a program to SSA form and maintaining it through various transformations can be complex and computationally expensive. Additionally, introducing $\phi$ functions can make the intermediate representation more difficult to handle, especially for large and complex codebases.

### 3.2.5 E-graphs and Equality saturation

Although e-graphs were originally developed in the late 1970s for use in automated theorem provers, a more recent technique known as equality saturation repurposes e-graphs to implement rewrite-driven compiler optimizations and program synthesizers [60].

The core idea of equality saturation is to represent a program in an intermediate representation and repeatedly apply a set of equivalence-preserving transformations until no further transformations can be applied. The goal is to explore all possible equivalent versions of the program within a certain space to find the optimal one according to some criteria, such as performance or simplicity [61]. The process typically involves three main phases. Extraction: Convert the original program into an intermediate representation. Saturation: Apply equivalence transformations to the intermediate representation iteratively; each transformation produces a new, equivalent form of the

program, which is added to the representation. Extraction: After the saturation phase, extract the best version of the program from the intermediate representation based on the optimization criteria.




(a) Initial e-graph contains $(a \times 2)/2$.

(b) After applying rewrite $x \times 2 \rightarrow x \ll 1$.

(c) After applying rewrite $(x \times y)/z \rightarrow x \times (y/z)$.

(d) After applying rewrites $x/x \rightarrow 1$ and $1 \times x \rightarrow x$.

**Figure 3.6:** Applying rewrites to an e-graph adds new e-nodes and edges, but nothing is removed. Expressions added by rewrites are merged with the matched e-class. The resulting e-graph has a cycle, representing infinitely many expressions: $a, a * 1, a * 1 * 1$, and so on. Taken from [60].

Advantages of equality saturation are: Comprehensive Exploration, by representing multiple equivalent forms, it can find optimal transformations that traditional optimization techniques might miss; Compact Representation, E-graphs allow for an efficient representation of many equivalent expressions, reducing redundancy; and Flexibility, it can be applied to various domains, including compiler optimizations, program synthesis, and theorem proving. Known disadvantages are: Complexity, computationally intensive for large programs or complex transformation sets; Termination, ensuring that the saturation process terminates can be challenging, as the space of possible transformations can be very large; and Extraction Difficulty, extracting the optimal program from the saturated e-graph can be non-trivial and might require sophisticated heuristics.

**Egglog.** Equality saturation was enhanced in 2023 by [62] when it was unified with Datalog, another reasoning framework with many applications, extensions, and high-quality implementations [63]. Datalog is well-studied by the databases community, and practitioners use it to build program analyses. They share a common setup: the user provides rules and an initial set of facts (a term in EqSat and a database in Datalog), and then the system derives a larger and larger set of facts from those inputs. Originally developed for Rust, it has a recent Python wrapper.

## 3.3 Approach

### 3.3.1 Framework

We propose two approaches to address the downstream task of predicting execution time: coarse-grained and fine-grained training strategies. The former performs training directly over a whole dataset, while the latter employs a hierarchical approach where the same dataset is divided into clusters, and individual cluster-wise regressors are trained and composed in a single model. Both approaches can be treated as black or white boxes by the user at inference time.

Both approaches are divided into three stages: (1) pre-training or data pre-processing, (2) training, and (3) inference. The first stage includes three tasks: code-to-graph conversion, graph-to-vector embedding, and database generation. The second stage is in charge of training all nine algorithms considered. Lastly, the third stage corresponds to inference, where unseen inputs are given to a black box (highlighted in blue in Figures 3.7 and 3.8), and predictions are obtained.

**Coarse-grained approach.** All combinations of available programs, code-to-graph methods, embedding algorithms, hardware architectures, problem sizes, workload, and performance metrics are gathered into the task of database generation. Then, six off-the-shelf learning algorithms are executed on train-validation-test subsets; the metric we aim to minimize is the NRMSE.

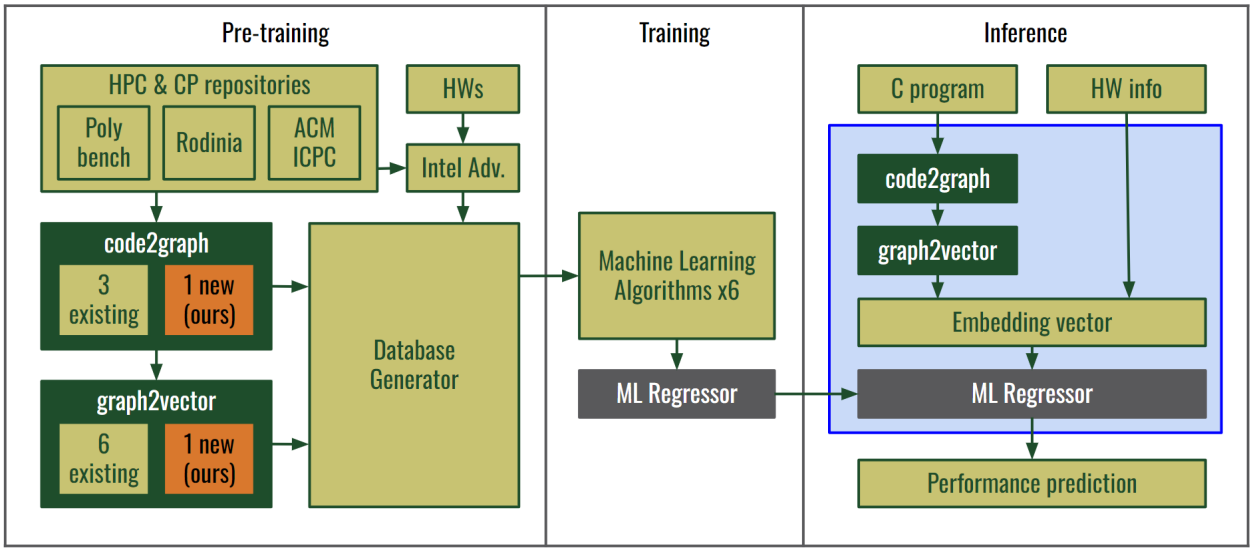



**Figure 3.7:** High-level system overview: coarse-grained approach.

**Fine-grained approach.** The task of graph-to-vector conversion is executed following the approach suggested by [59] for graph classification, "to learn the embedding of each graph by treating them as a subgraph of a union of all the graphs." Instead of using a pooling operator as in the coarse-grained approach, we used meta-nodes connected to all nodes inside each subgraph.

Training is divided into three sub-models. First, all input instances are clusterized using K-Means based on performance metrics such as execution time, GFLOPS, and memory (RAM + caches) bandwidth. The idea is to group programs with similar performance behavior together.

Second, a classifier is trained with Sub2Vec embeddings as inputs and clusters as outputs. Lastly, one regressor is trained per cluster, aiming for lower NRMSEs than the coarse-grained approach due to the similarity of program behavior inside each cluster.

Inference can be seen as a black or white box (highlighted in blue in Figure 3.8). Per input, only two models are executed, out of the $K + 1$ models obtained, where $K$ = #clusters. One will classify the input program into a cluster based on its embedding vector; the other will predict the execution time based on other input features such as HW specs, workload, and problem size.

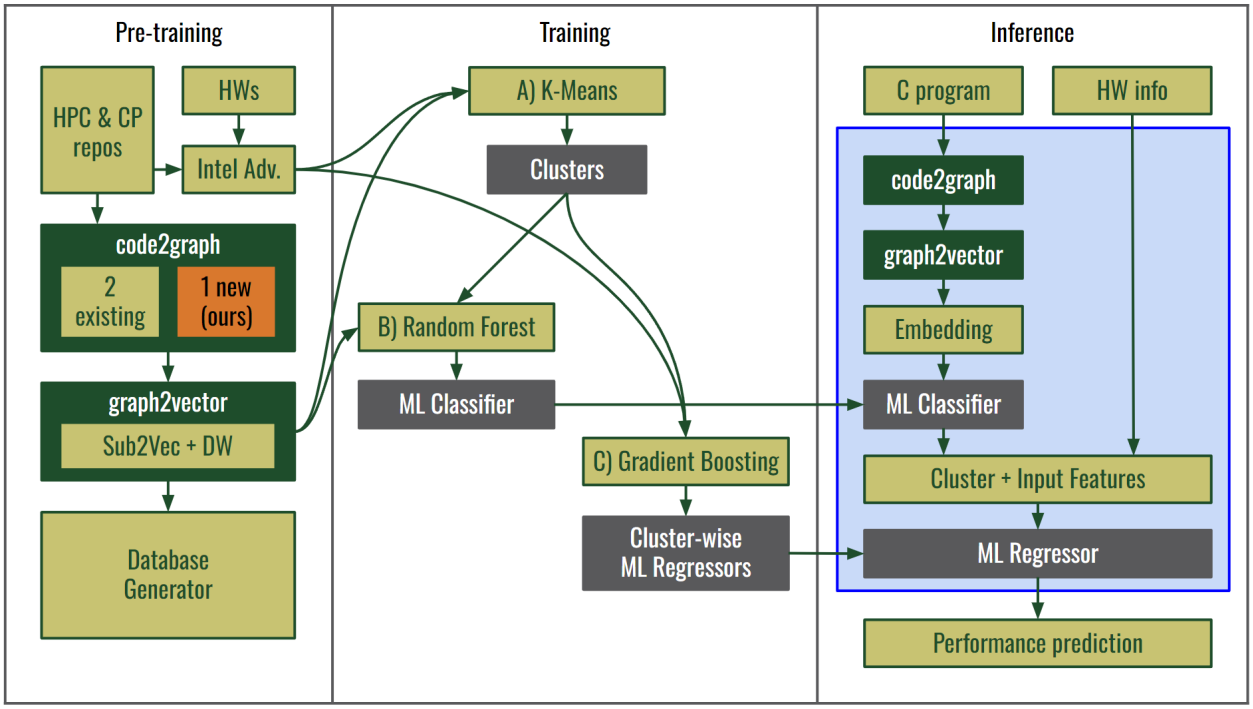



**Figure 3.8:** High-level system overview: fine-grained approach.

### 3.3.2 Implementation

The majority of the code was written in vanilla Python, along with several other libraries. However, database generation required a complex orchestration of several machines concurrently executing bash scripts, compiling C code, performing Intel Advisor's analysis, running Python parsers, and gathering the results in a centralized GitHub repository.

It is important to mention that execution time and employed resources differ dramatically between stages. Database generation took around two months to execute and required more than seventeen machines from the CS Department of CSU. Training took a day and a half for the coarse-grained and around 12 hours for the fine-grained approach, running into the "bonito" and "wahoo" machines with 6GB of GPU. Lastly, the average inference time is 0.407ms per unseen input.

## 3.4 System Overview

### 3.4.1 Egg-no-graph: Design of our code to graph representation

Starting from an SSA, an egg-no-graph can be understood as an e-graph, where e-nodes represent basic blocks (BBs), variables, operators, instructions, functions, and procedures. A type system is needed to ensure compatibility in the e-graph, implemented as "classes" and "functions" in Egglog [62]. Edges represent flow, both inter-BBs (among BBs) and intra-BBs (among instructions inside them). Last but not least, whenever a function is called, we add a pair of edges to and coming from between the caller and the function's initial and final basic blocks.

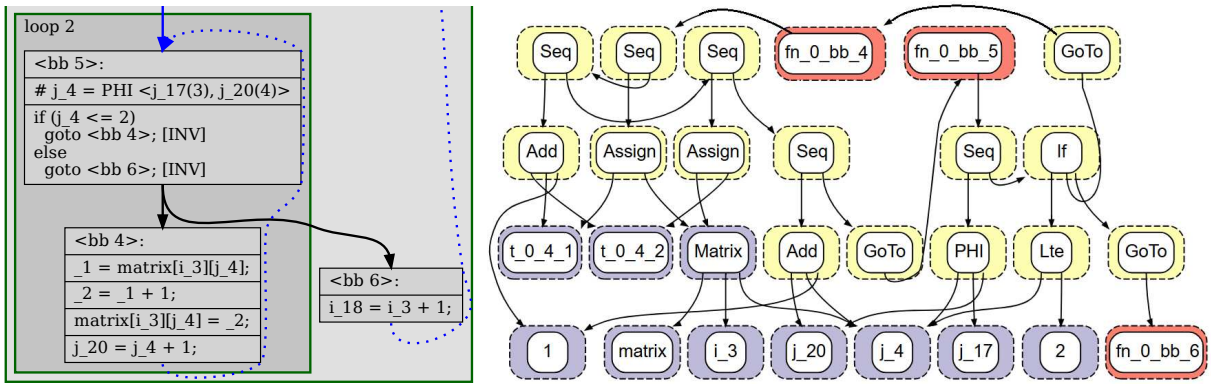



**Figure 3.9:** Egg-no-graph representation or an equality saturated version of a given SSA (partial).

There are four classes: Basic Block, Variables, Operations, and Functions. In Figure 3.9, we used a color code to distinguish each class type. E-nodes colored in red are Basic Blocks, yellow represents Operations, purple represents Variables, and a fourth color designates Functions; due to Figure 3.9 being an example of a subset of an SSA which does not contain “entry” or “exit” basic blocks neither calls to Functions or Procedures, thus the fourth color is not shown.

One of the most remarkable features of egg-no-graph is its creation process, which has linear complexity over the content of a given SSA graph. This means that each SSA feature is visited only once, eliminating the need for backtracking and ensuring efficient creation performance. The reader can find all the algorithmic details at the **Appendix A** of this document.

The vanilla version of egg-no-graph stores constant values as numbers. This way, if a particular value is utilized in different basic blocks of the same function, it will be stored in the same e-class. We implemented a variation where each value is stored abstractly, mimicking how temporal values are handled in SSAs. This means using a generic name instead of its actual value. Note: both versions contain the same number of edges, but the modified one possesses more nodes.

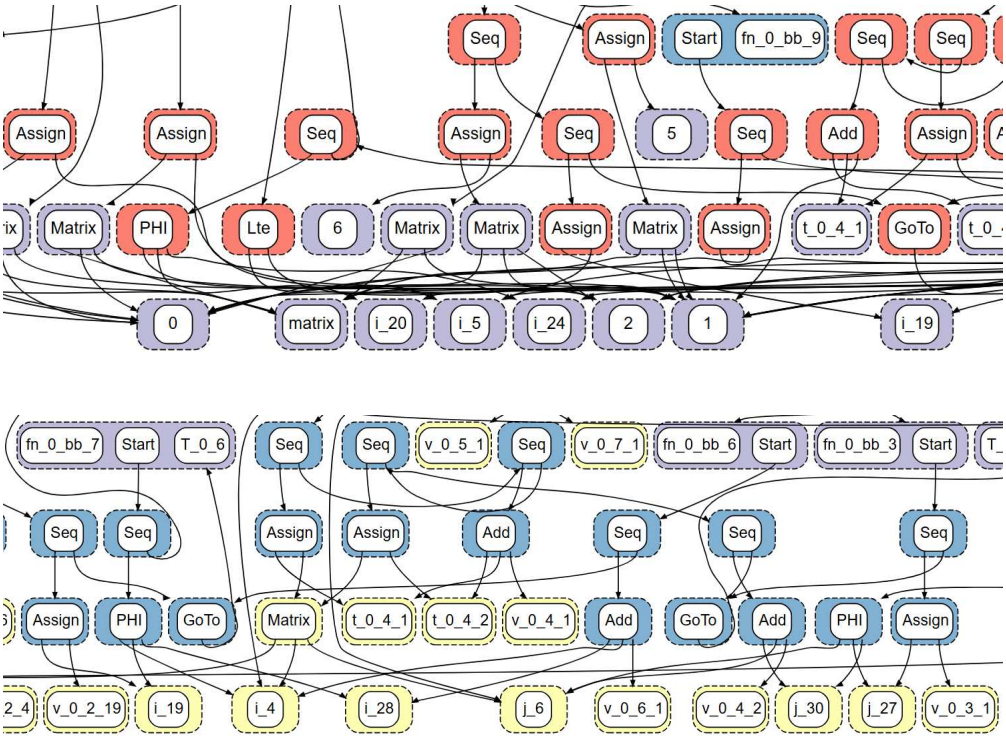



**Figure 3.10:** Partial view of Vanilla (top) and modified (bottom) versions of Egg-no-graph.

### 3.4.2 Bag of Colors: Design of fast WL-based graph embedding.

The 1-WL color refinement algorithm can be extended to any amount of graphs; this is known as the W-L kernel where graphs are represented as bag-of-colors. It is computationally efficient and closely related to GNNs, as it enriches node colors by applying K-step color refinement algorithm; thus, different colors capture different K-hop neighborhood structures.

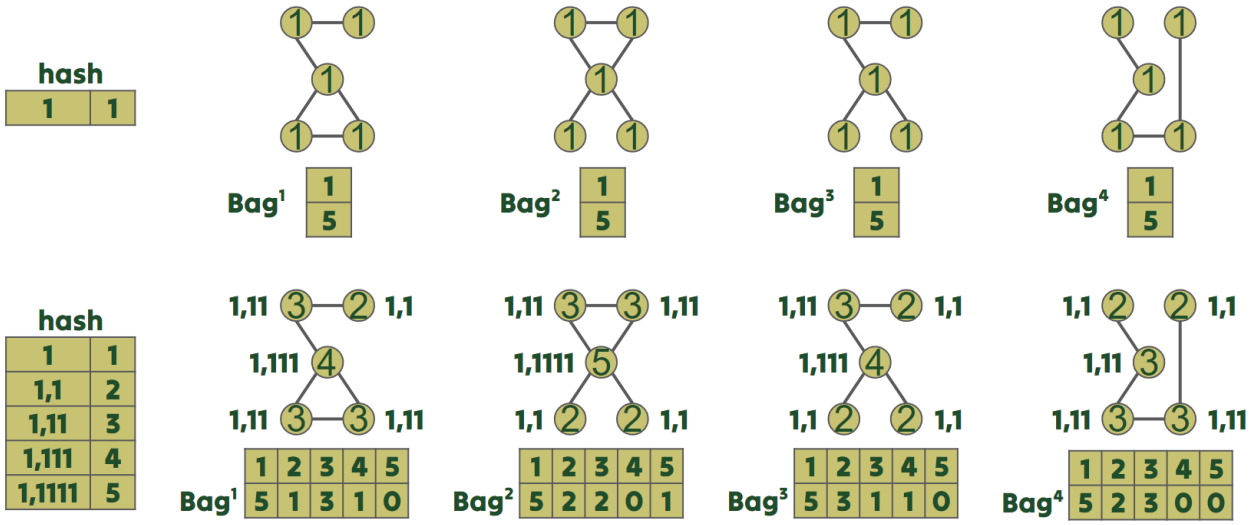



**Figure 3.11:** First and second iterations of the W-L kernel applied to four graphs.

It is capable of creating graph-level embedding vectors efficiently, as its time complexity per iteration is linear with the number of edges, and its spatial worst-case scenario will be linear with the number of nodes (one color per node). Its major disadvantage, though, comes when using the vectors in machine learning models because some colors can be very sparse when the amount of graphs or iterations is large. Example: in Figure 3.11, color 5 only appears in 1 out of 4 graphs; in the next iteration, new colors that contain color 5 in their definition will be even less frequent.

Our approach is aware of this sparsity, and we address this issue by removing (from the bag) those colors with lower standard deviation than a given threshold, thus keeping only those features with more information content. Typical values for the threshold range from 0.5, 1.2, or 1.7; it will depend on the specific graph dataset given as input. The higher the value, the more strict the color filtering is. Thus, more iterations will be needed for the algorithm to finish.

**Algorithm 2** Bag of structural colors graph embedding

**Require:** S $\leftarrow \{G_1, G_2, G_3, \ldots G_n\}$, threshold, $k \leftarrow$ vector length ▷ Input graphs
**Require:** $G_i = (V_i, E_i) \forall G_i \in S$ ▷ Structural init
hash $\leftarrow (1, 1)$ ▷ Only one in general, all graphs share the same hash table
$c_v^0 \leftarrow \text{hash}[1] \forall v \in V$ ▷ One per graph. $|c|$ = #iterations × #nodes
$\text{Bag}^i_{\text{hash}[1]} \leftarrow |V_i| \forall G_i \in S$ ▷ One per graph. $|\text{Bag}^i|$ = #colors
**while** $|\text{Bag}^n| < k$ **do**
    $t_v \leftarrow \text{aggregate}(c_v^{l-1}, \{c_w^{l-1} : w \in N_G(v)\}) \forall v \in V$
    $\text{hash} \leftarrow (t_v, |\text{hash}| + 1) \forall v \in V \wedge t_v \notin \text{hash}$
    $c_v^l \leftarrow \text{hash}(t_v) \forall v \in V$
    $\text{Bag}^i_x \leftarrow |(c_v^l = x | \forall v \in V)| \forall x \in l$
    $\text{Bag}^i = \text{Bag}^i - \text{Bag}^i_x |\ \text{stdev}(\text{Bag}_x) <$ threshold ▷ Received as function parameter
**return** $\text{Bag}^i_{[1,k]} \forall G_i \in S$

We propose an improvement of this algorithm by changing the initialization of the colors. Instead of using a constant value or a structural feature such as node degree, we can leverage node types. We call this improvement a "bag of semantic colors" and it is particularly useful with heterogeneous and/or multipartite graphs (like e-graphs, which have a type system).

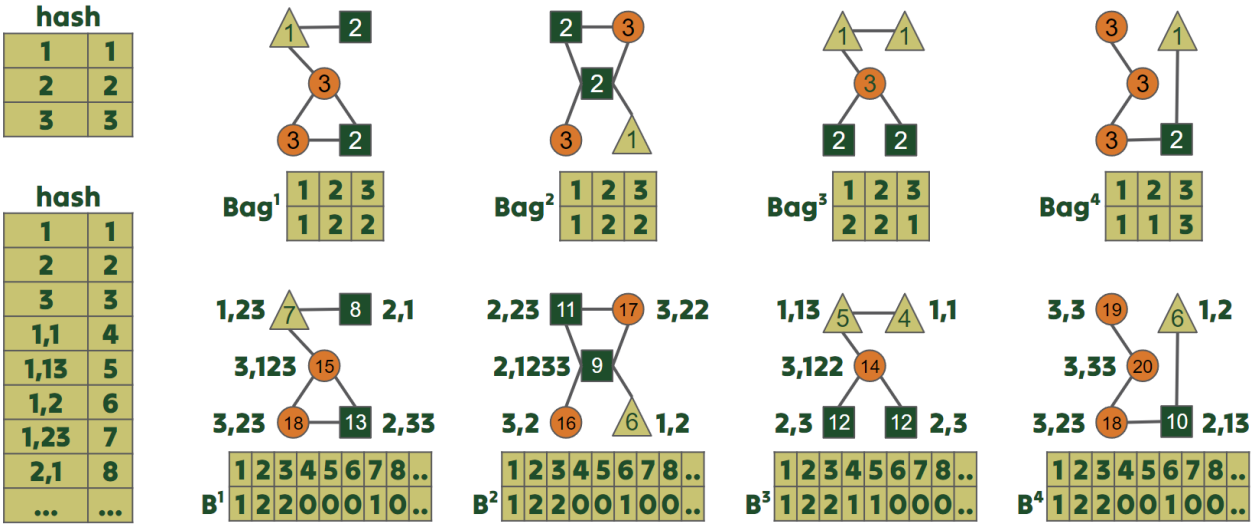



**Figure 3.12:** Bag of semantic colors: 1st and 2nd iterations applied to four heterogeneous graphs.

Introducing types leads to a higher number of colors produced per iteration (since more combinations can be obtained); this can be addressed by relaxing the threshold or using shorter vectors. More iterations are generally better since they increase the K-hop neighborhood captured.

### 3.4.3 Design of a heterogeneity-aware data set

We aim for our dataset to offer trainable instances not only to the model discussed in this work but also a wide design space of other models related to graph embedding and performance characterization. The core of our dataset is a set of scientific programs originally written in C/C++.

There are three dimensions of heretogeneity in our experiments. Firstly, each program will process different problem sizes, thus offering variety in execution time, flops, and memory bandwidth requirements. Secondly, each program will be executed on multiple machines with different specifications. Lastly, each machine will be asked to execute the benchmark multiple times on different days of the week, thus offering a variety of CPU and memory usage, "workload."

From the embedding generation perspective, we aim to obtain all combinations of two factors: code-to-graph representations and graph-to-vector embedding algorithms. We included as many kinds of embedding approaches as possible inside this work's limitations: random walk-based, graph kernels, proximity reconstruction-based, DNN-based, and spectral-based.

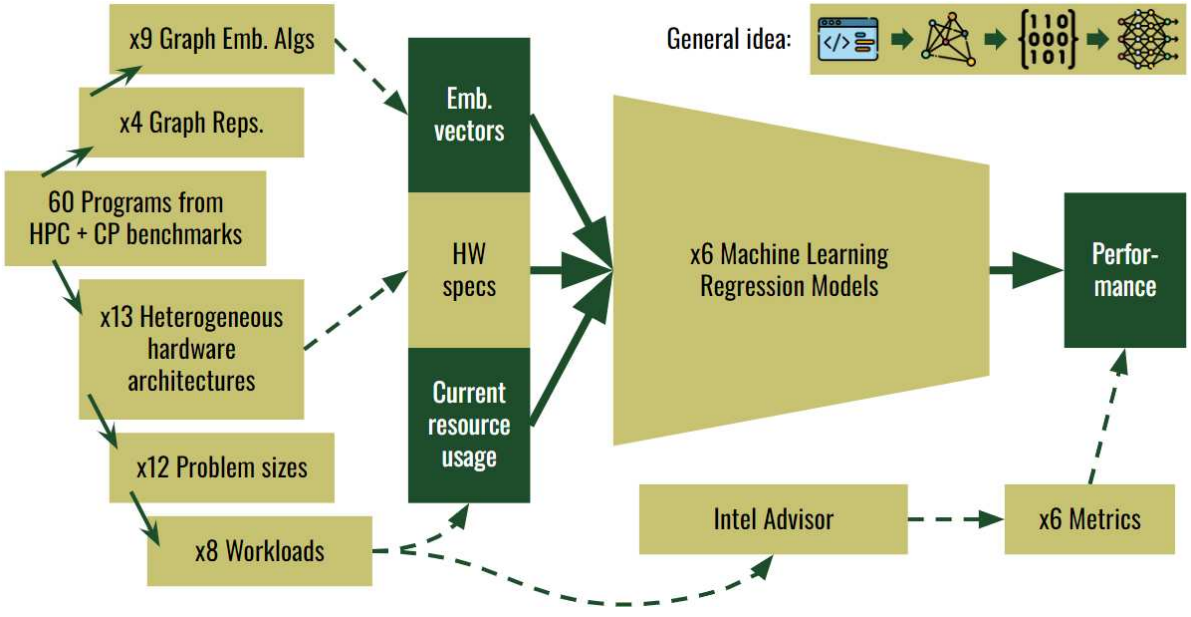



**Figure 3.13:** High-level overview of our data set's methodology.

For this work, we focus only on execution time, but more metrics can be explored in future work, like GFLOPS and memory (RAM and cache) bandwidth, as provided by Intel Advisor.

## 3.5 Implementation and Evaluation

### 3.5.1 Task: Code-to-graph conversion

Our first task consisted of choosing which programs to use as our dataset's core. Three widely known repositories in the HPC and CP communities were selected for a total of 60 programs.

- **PolyBench** is a benchmark suite of 30 numerical computations extracted from operations in various application domains like linear algebra computations, image processing, physics simulation, dynamic programming, statistics, etc. [64].
- **ACM-ICPC** (Association for Computing Machinery–International Collegiate Programming Contest) is a worldwide annual multi-tiered programming contest sponsored by IBM. It brings together 50,000+ students worldwide per year from 3,000+ universities in 111 countries. Typical topics include data structures, sorting, dynamic programming, backtracking, number theory, combinatorics, and greedy, graph, geometric, and network flow algorithms.
- **Rodinia**, a benchmark suite for heterogeneous computing. Rodinia includes applications and kernels that target multi-core CPU and GPU platforms. According to its authors [65], Rodinia covers a wide range of parallel communication patterns, synchronization techniques, and power consumption. In this work, only a subset of Rodinia's programs are used.

We selected a total of four graph representations, finally stored in EasyGraph's format.

- **Abstract Syntax Tree:** obtained using gcc's preprocessing tool and PyCParser library [66].
- **Static Single Assignment:** obtained with the help of gcc's fdump tree "all graph" tool.
- **HARP:** is an UCLA end-to-end graph-based solution [67] for modeling HPC programs for FPGA-based synthesis. It builds on top of (augments) GNN-SDE and ProgramL; refer to Related Work for more details. A subset of 21 graphs of PolyBench programs is utilized.
- **Egg-no-graph:** handcrafted Python functions spanning 456 lines of code using the Egglog library, taking SSA.dot files as input and generating EasyGraph objects as output. For further details, please check the Egg-no-graph design in the "System Overview" section.

### 3.5.2 Task: Graph-to-vector embedding

Once all the programs are converted to the graph representations detailed in the previous section, the next task is to embed them in vectors. We utilized seven graph embedding methods:

- **Deepwalk and node2vec:** Based on random walks, these methods work at the node level and obtain vectors by learning node occurrences on pseudo-random (biased) sequences.
- **LINE:** Another shallow embedding approach, it learns 1st and 2nd-order proximity of nodes.
- **SDNE:** It uses an autoencoder (deep neural network) to learn a graph's adjacency matrix.
- **NOBE and NOBE-GA:** Focus on transforming graph data to the spectral domain; their learning is based on the graph Fourier transform. Don't escalate well with vector length.
- **Bag of Structural Colors:** Our version of the W-L kernel is the only method that operates at the graph level. Similar in concept to a message-passing graph neural network.

We utilized the EasyGraphs library to obtain node-level features, and then we used average as the pooling operator for all methods that need one. Vector lengths explored range from 2 to 256.

Execution time-wise, our implementation beats other methods by far. LINE and SDNE show the longest execution time. NOBE and its approx. calculate short vectors fairly quickly but do not converge in longer ones. Eggnog takes longer time than other graphs due to larger node amounts.

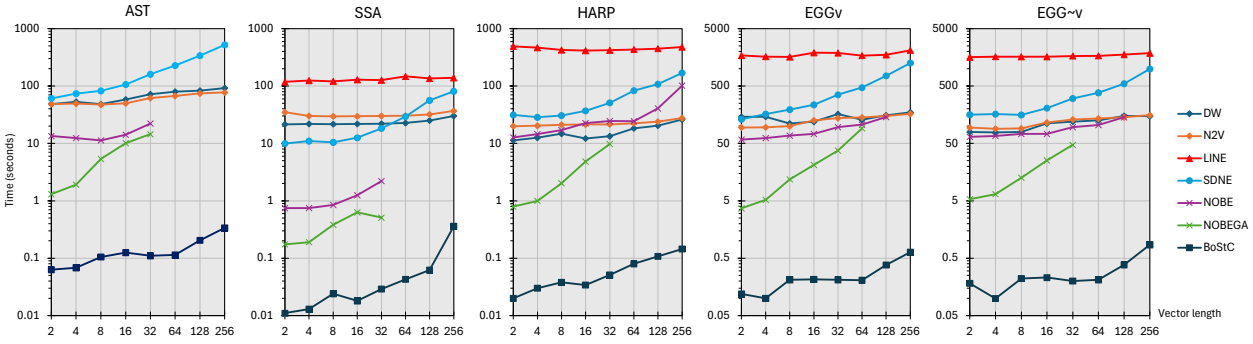



**Figure 3.14:** Execution time of each combination of graph representation and embedding method (lower is better in the vertical axis). The horizontal axis represents vector lengths. **Note: Figure in log-log scale.**

### 3.5.3 Task: Database generation

Our database contains a total of 150M+ training instances, calculated as follows: 60 Programs x 13 HWs x 12 Problem sizes x 8 Workloads x 6 Performance Metrics x (5 Graph Reps. x 9 Emb. Algs. x 8 Emb. Sizes - 30 divergences) = 449,280 x 330 = 148,262,400 unique instances.

The remaining 2M+ instances were obtained during the fine-grained approach, where 3 Graph Reps. x 8 Emb. Sizes x 21 Programs + 4 Emb. Sizes x 60 Programs = 744 were added to existing 13 HWs x 5 Problem sizes x 8 Workloads x 6 Performance Metrics = 3,120. Resulting in additional 744 x 3,120 = 2,321,280 instances. **Adding up to 150,583,680 instances in total.**

We are assuming one process at a time. While we are aware that generating training data on shared machines may lead to errors, it is future work to extend to shared-memory multiprocessing.

Our high-level goal is to ensure generalizability across four dimensions: new hardware architectures, unseen programs, and resource usage in diverse circumstances, such as problem sizes and workloads (memory and CPU usage) while answering Research Questions 3, 5, and 6.

When using the whole database, we suggest the train-validation-test distribution of Table 3.1.

**Table 3.1:** Distribution of independent variables among train-val-test subsets.

| Variable | Total | Training | Validation | Testing |
|---|---|---|---|---|
| Hardware | 13 | 9 | 2 | 2 |
| Programs | 60 | 52 | 4 | 4 |
| Problem Sizes | 12 | 8 | 2 | 2 |
| Workloads | 8 | 4 | 2 | 2 |

#### Sub-task: Data subsets for experiments

Each experiment focused on a particular subset of our database, detailed as follows.

- **Dataset for Experiment #1: coarse-grained training | Eggnog vs. baselines.**

  **Dataset for Experiment #2: fine-grained training | Eggnog vs. baselines.**

  We are limited to 21 programs due to state-of-the-art baseline HARP constraints.

- **Dataset for Experiment #3: fine-grained training | Eggnog.**

  We used all 60 available programs: Polybench + ACM ICPC + Rodinia.

### 3.5.4 Task: Experimental results #1: coarse-grained training

After a dense grid of models was trained on the dataset, mixing all input combinations of graph reps and embedding methods, our experimental results are shown in Figure 3.15. We utilized TensorFlow on the "bonito" machine of the CS department, whose specifications are HP-Z440-XeonE5-1650v4, 6x3.6Gh, 16Gb RAM, and 6GB GPU.

Our graph representation, Egg-no-graph, in its variant (blue), offers the smallest normalized root mean squared error (NRMSE) compared to other graph reps on average. In second place, with almost the same average NRMSE we found Egg-no-graph (green) and AST (red). To our surprise, HARP (black) presents the highest NRMSE on average.

Embedding method-wise, Figure 3.15 right, the best methods are node2vec, LINE, Deep-walk, and NOBE-GA. It is encouraging to see that Bag-of-colors performs similarly to NOBE and SDNE, though it is calculated in much less time.

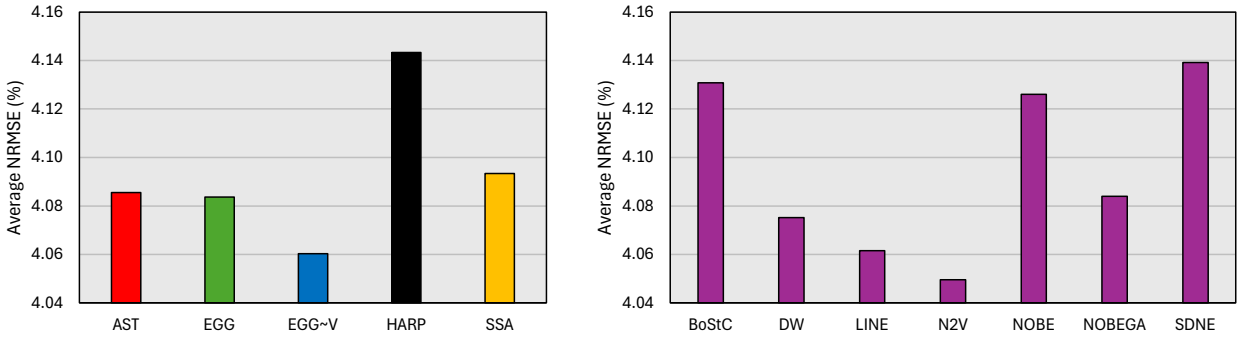



**Figure 3.15:** Normalized root mean squared error graph rep-wise (left) and embedding method-wise (right).

Going deeper into how each graph rep performed on each embedding method, Figure 3.16, we find that Egg-no-graph performs consistently among methods. HARP performs best on Bag-of-colors and worst on SDNE. AST performs best on node2vec and worst on Bag-of-colors. It is interesting to note that Bag-of-colors performance varies dramatically with input.

The lesson we learned is that no embedding method performs best in all cases; it is a parameter we should explore when dealing with future GraphML problems. On the other hand, SSA and Egg are consistently better among all methods, while AST and HARP's accuracy varies the most.

This can be quantified and seen in Figure 3.16 (right), where the standard deviation for Egg-no-graph is lowest among other graph reps, while AST and HARP's standard deviation is highest.

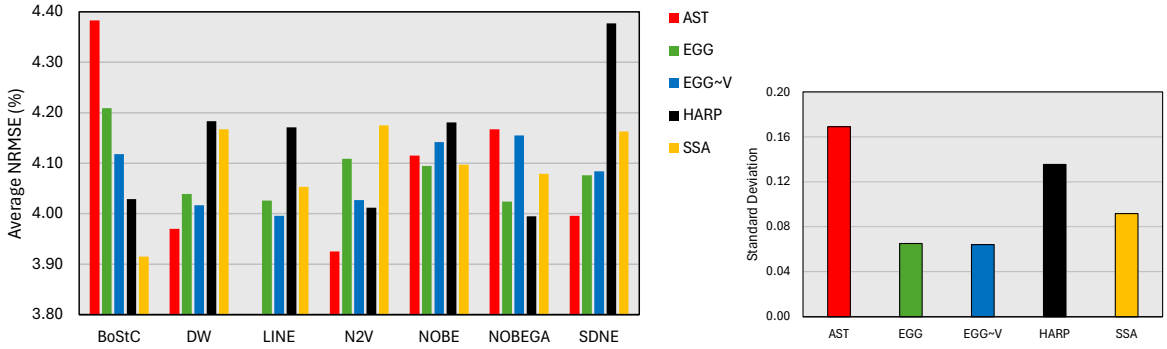



**Figure 3.16:** Detail of normalized root mean squared error per each embedding and graph combination (left) and graph rep-wise standard deviation (right). Lower is better in both plots.

Lastly, taking a look at Figure 3.17 gives us a valuable insight: not all machine learning methods perform similarly. The best one by far is software vector machines (SVM), consistently giving the best performance independently of the graph rep. The second place is neural networks. AdaBoost consistently reported the highest error rates. Egg-no-graph variation gives the best result on 4 out of 6 machine learning models, equal result on SVM, and is outperformed only on AdaBoost.

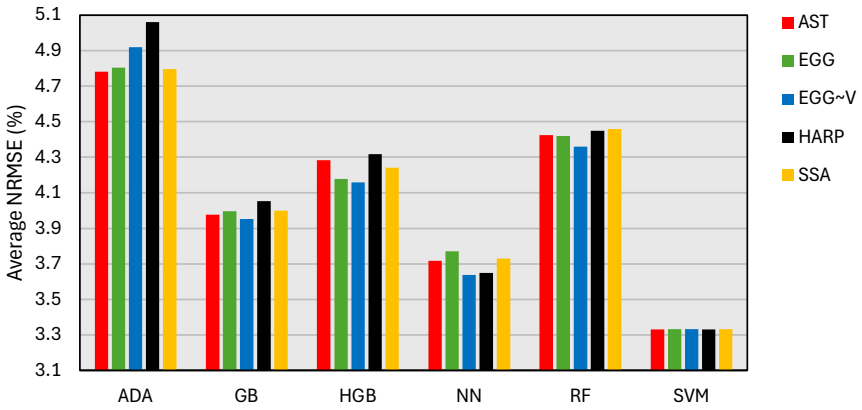



**Figure 3.17:** Normalized root mean squared error per machine learning algorithm (lower is better).

We found no clear trend in NRMSE vs. embedding vector size, reason why it is not reported.

### 3.5.5 Task: Experimental results #2: fine-grained training

Using the same experimental setup, three executions were obtained per each combination of graph representations (egg-no-graph, HARP, and AST), embedding sizes (N = 2, 16, 128), and clusters (K = [2,25]). Averages are shown in Figure 3.18. Not only does our approach outperform other graph reps on all embedding sizes, but it also reaches NRMSEs below 4%, the best result of Experiment #1. Regarding vector lengths, small lengths report better accuracy on K between 2 and 9 (peaking at K = 5); longer vectors behave consistently better on clustering above K = 10.

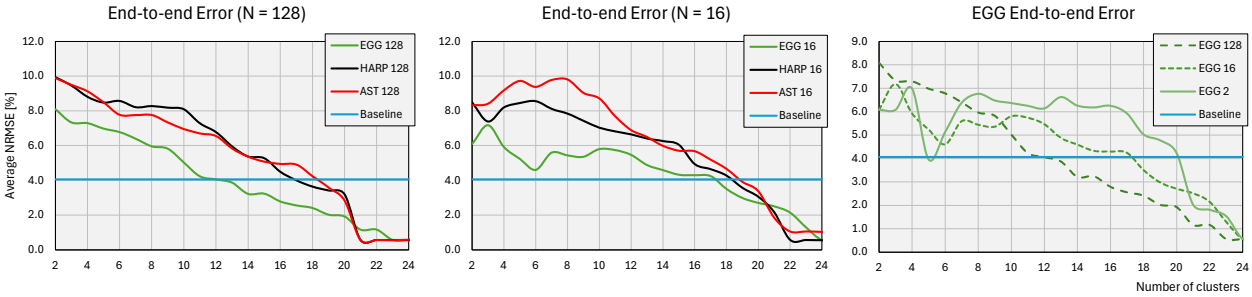



**Figure 3.18:** End-to-end NRMSE of Eggnog vs. HARP vs. AST for embedding vector sizes N=128 (left), N=16 (center), and N=2 (right). Baseline is the best result of Experiment #1. Lower is better in all plots.

Considering that our fine-grained training approach is divided into three stages, it is important to report egg-no-graph's performance against baselines on each stage. Due to obtaining the best results in Figure 3.18 and for brevity, we focus on reporting results for embedding size N = 128 in Figure 3.19. It is important to note that the merging point on all graphs is K = 21, where the amount of clusters equals the number of programs in the subset. Other Ns behave similarly.

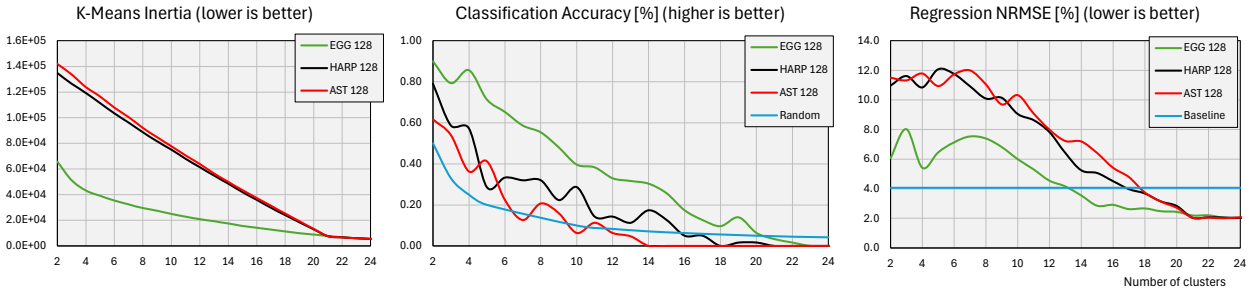



**Figure 3.19:** Experiment #2's partial performance comparison. Egg-no-graph outperforms baselines in all sub-models: clustering inertia (left), classification accuracy (center), and regression error (right).

### 3.5.6 Task: Experimental results #3: fine-grained training

In our last experiment, we explore eggnog's behavior while training it over all 60 available programs for different embedding sizes; we report a comparison, Figure 3.20, between end-to-end NRMSE and overall training time. Both metrics appear to be proportional to vector length. Each data point is an average of four independent executions ran on the "wahoo" machine of the CS Dept., whose specifications are identical to those of "bonito," used in Experiments #1 and #2.

Previously, we obtained the Sub2Vec-based DW Eggnog vectors for all 60 programs; average times were: 0.485 (N=2), 0.495 (N=16), 0.898 (N=128), and 1.075 (N=256) seconds per program.

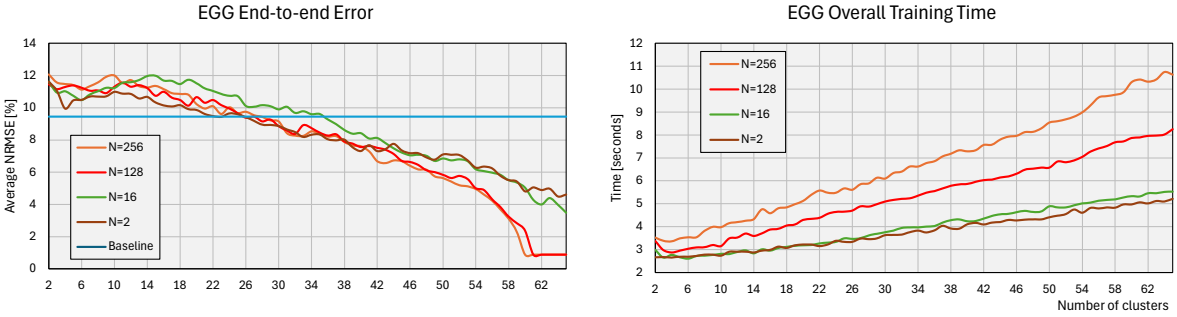



**Figure 3.20:** Experiment #3's main results: Eggnog end-to-end NRMSE while fine-grained trained over all 60 available programs for embedding sizes N = {2, 16, 128, 256} (left) and overall training times (right). Baseline is the best result of CS793 poster, where eggnog was coarse-grained trained with all 60 programs.

Finally, we report the training times per each stage in the fine-grained trained model: K-Means clustering, Random Forest classifying, and Gradient Boosting regression are shown in blue, orange, and green colors, respectively, in Figure 3.21. Regression represents more than half the overall time on N = 2, but remains constant; and it is surpassed by other sub-tasks on longer vector sizes.

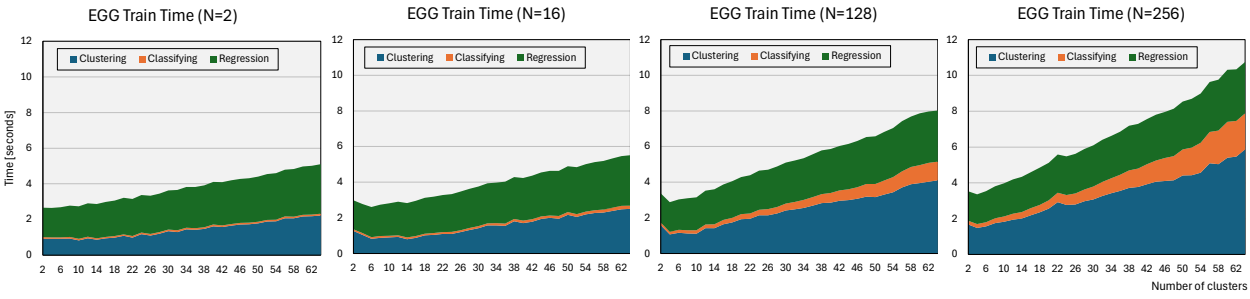



**Figure 3.21:** Detail of egg-no-graph training time per stage, per different embedding sizes.

## 3.6 Conclusion

First, we introduced a novel code-to-graph representation based on equality saturation. Our approach utilizes e-graphs as its backbone. Second, we augment a graph kernel embedding method, making it suitable for use in machine-learning environments. Its training time is orders of magnitude faster, while its accuracy is similar to that of other algorithms, making it suitable for time-critical systems. Lastly, we designed and created a dataset with 150M+ instances, mixing multiple programs, hardware architectures, performance metrics, code-to-graph, and embedding approaches. Our dataset can be easily escalated to suit future research project needs.

## 3.7 Future Work

- We look forward to finding out why the modified Egg-no-graph performs better on average than the vanilla version. Also, design more variations and add languages (Fortran or Python).
- Provide a deeper characterization of the minimum number of clusters needed to outperform coarse-grained modeling. Specifically, where baseline intersects on Figures 3.18 and 3.20.
- Compare our approach to other graph representations, such as PerfoGraph [68] and PrograML [69]. Also, either broaden our program's pool or restrict it to a particular domain, such as ML-related applications. Find a tradeoff between accuracy and generalizability.
- Explore additional pooling operators other than average for node-to-graph-level conversion.
- Explore the usage of Sub2Vec's approach with other embedding algorithms rather than Deepwalk; particular focus will be given to our method, Bag of Colors.
- Explore vector combinations and concatenations; this means combining more than one approach into a single vector, perhaps guided by t-SNE visualizations.
- Implement the remaining Bag-of-Semantic-Colors and Graphlet kernels methods.
- Guide our research to a more verifiable path than the trial-and-error nature of traditional machine learning algorithms; we plan to explore the usage of formal methods in Graph ML.

## 3.8 Related work

In the last decade, since the publication of DeepWalk in 2014 [42], thousands of projects worldwide have leveraged Graph ML and graph neural networks to tackle a wide variety of downstream tasks; see Figure 3.22. Researchers from diverse fields have modeled all kinds of input data as graphs to solve specific problems. The remarkable growth and success of Graph ML has slowly expanded its applications into programming languages and program analysis. However, a key challenge in adopting the latest ML methods is the representation of programming languages, which directly impacts the ability of AI to reason about programs [68].

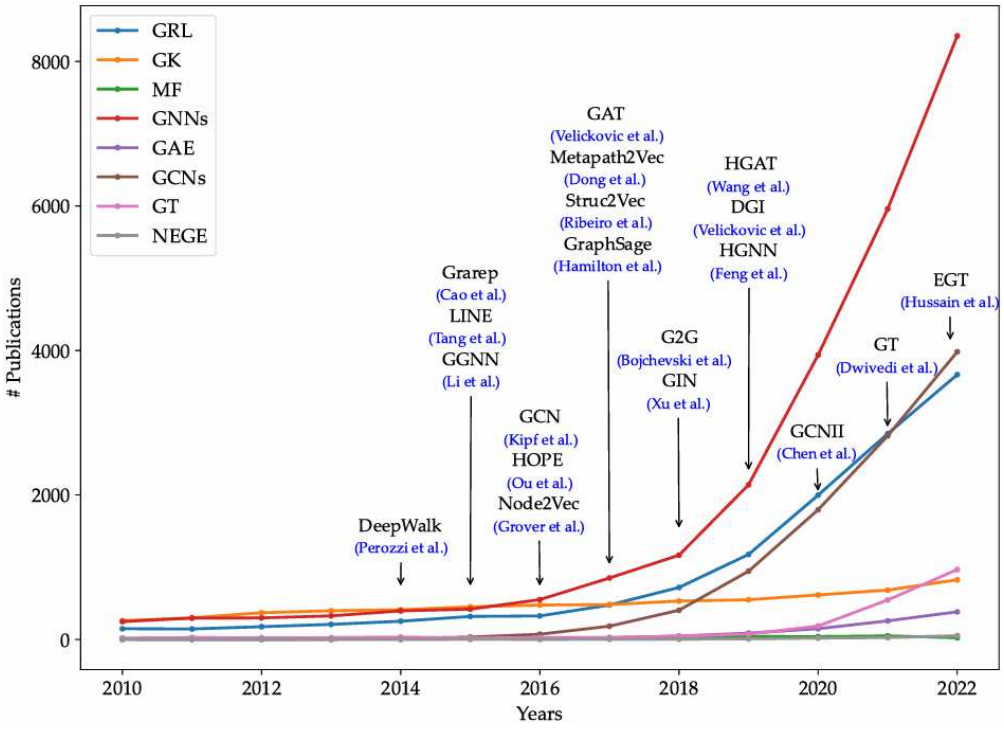



**Figure 3.22:** Popularity of graph representation learning models in the Scopus database. Taken from [38].

**Where it all started: “PrograML" Graph-based Deep Learning for Program Optimization and Analysis. It is the baseline for the next papers.**

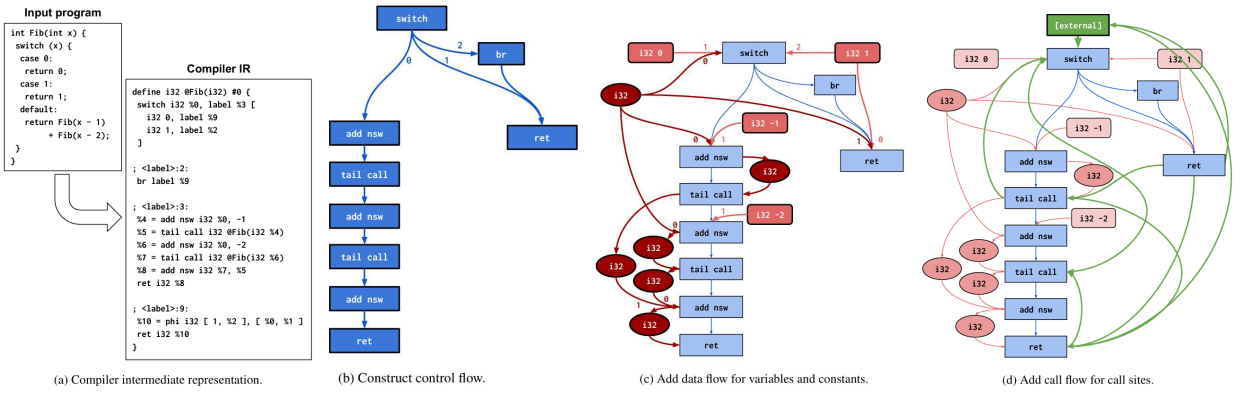
Even though machine learning offers significant benefits for constructing optimization heuristics, further research on two key ideas is needed to achieve the performance of an optimal heuris-

tic. These key areas are a representation that accurately captures the semantics of programs and a model architecture with sufficient expressiveness to reason about this representation [69].

ProgramML stands for Program Graphs for Machine Learning; it is a graph-based program representation that uses a low-level, language-agnostic, and portable format designed for use in machine learning models to perform downstream tasks over these graphs. ProgramML provides a general-purpose program representation that equips learnable models to perform the types of program analysis that are fundamental to optimization.

By the time of its publication, ProgramML supported just LLVM and XLA IRs, but it currently adds support for C and C++ code. It is implemented as a directed attributed multigraph. Message-passing neural networks propagate information through this structured representation, enabling whole-program, per-instruction, and per-variable classification tasks.

The motivation behind ProgramML is to replace fragile and expensive hand-tuned heuristics with data-driven statistical modeling in traditional compiler analysis tasks such as control flow reachability, dominator trees, data dependencies, variable liveness, and common subexpression detection [69]. A ProgramML graph G = (VE) is constructed by traversing a compiler IR. Graph construction is divided into three stages: control-flow, data-flow, and call-flow. The latest implementation achieves this in O(V +E) passes [69].

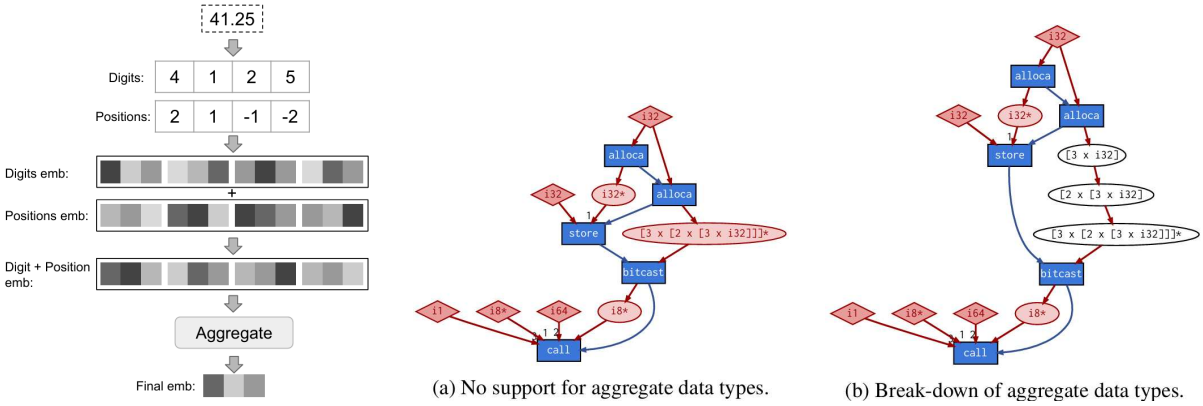



(a) Compiler intermediate representation. (b) Construct control flow. (c) Add data flow for variables and constants. (d) Add call flow for call sites.

**Figure 3.23:** Construction of a ProgramML representation for a simple C implementation of recursive Fibonacci using LLVM-IR. Taken from [69].

**"PerfoGraph," or how to augment PrograML by adding new functionalities based on number embeddings and aggregate data types.**

The authors proposed a graph-based program representation called PerfoGraph. The paper describes how PerfoGraph is used for applications such as program analysis, performance optimization, and parallelism discovery. PerfoGraph is a graph representation based on LLVM IR, built on top of PrograML. The author's claim is that despite PrograML's success, this program representation has shortcomings, especially for performance-oriented downstream tasks; PerfoGraph addresses those limitations by introducing new functionalities, such as number embeddings and aggregate data types, shown in Figure 3.24.




(a) No support for aggregate data types.

(b) Break-down of aggregate data types.

**Figure 3.24:** PerfoGraph's number embedding (left) and aggregate data types (right). Taken from [68].

**The need for GNN-aided Automated Accelerator Optimization or why GNNs and FPGAs, Field-programmable gate arrays, are friends.**

In recent years, Field-programmable gate arrays (FPGAs) have become an alternative for reconfigurable, energy-efficient, high-performance computing. According to [70], FPGAs are not yet widely adopted despite their potential advantages because of their complex programmability.

High-level synthesis (HLS) has freed computer architects from developing their designs in a very low-level language and needing to specify exactly how the data should be transferred at the register level. With the help of HLS, the hardware designers must describe only a high-level

behavioral flow of the design. However, it still can take weeks to develop a high-performance architecture, mainly because many design choices at a higher level require several minutes to hours to get feedback from the HLS tool on the quality of each design candidate.

Recent research has demonstrated that leveraging learning algorithms can mitigate this problem. Graph neural networks (GNNs) have been found to be highly effective in the electronic design automation (EDA) domain [71] [72] [73]. These works represent the input program or circuit as a graph and utilize GNNs to summarize the graph properties and produce a vector for graph/node embeddings. One of the main challenges is representing the HLS design (a C/C++ program with architectural pragmas) in a way that captures all relevant details and is informative for the learning model. It is common to utilize HLS as a black box and focus on developing efficient heuristics to explore the solution space efficiently [74].

**"GNN-DSE," or how to augment ProgramML with pragmas to explore design spaces and enable Automated FPGA Accelerator Optimization.**

[73] proposes an approach to solve this problem by modeling the HLS tool with a graph neural network (GNN) trained for a wide range of applications. The experimental results show that the GNN-based model is able to estimate the quality of design in milliseconds with high accuracy which helps search through the solution space very quickly.

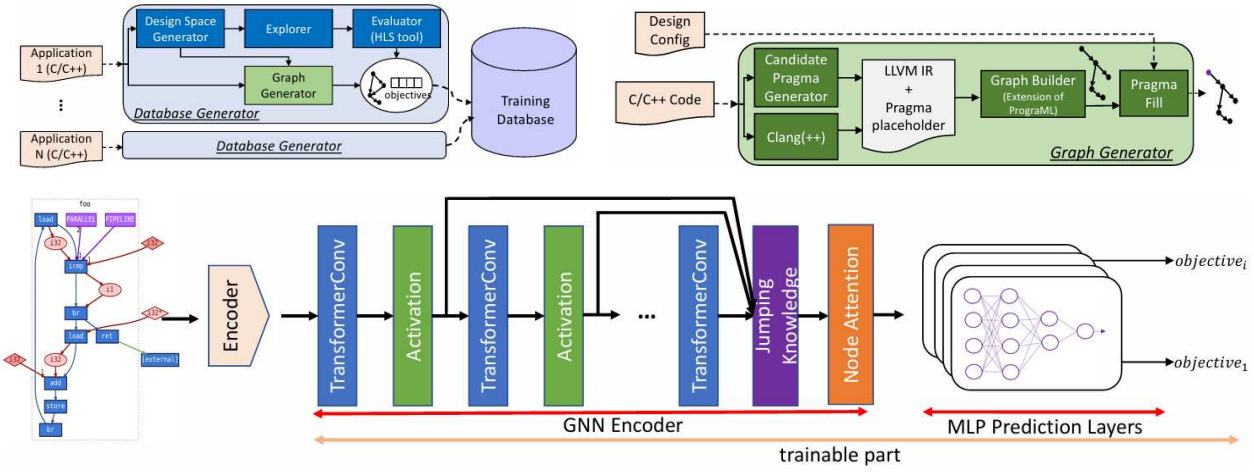



**Figure 3.25:** Architecture details of GNN-DSE: Database generator (top left), Graph generator (top right), and predictive model (bottom). Taken from [73].

GNN-DSE operates in three modes: training, inference, and DSE. During training, it takes its database as the input and trains a predictive model to learn to estimate the design's objectives until convergence. In the inference, GNN-DSE takes a C/C++ code and the desired design configuration as input and generates a graph representation. Then, it employs the trained predictive model to estimate the design's objectives in milliseconds. Lastly, the predictive model evaluates each design configuration during the DSE phase. As visualized in Figure 3.26, the GNN encoder successfully predicts configurations' latency and assigns embeddings to each graph so that designs with similar latency are clustered together. Once the exploration is finished, the top M designs are synthesized using the HLS tool to augment the database with their true objectives.

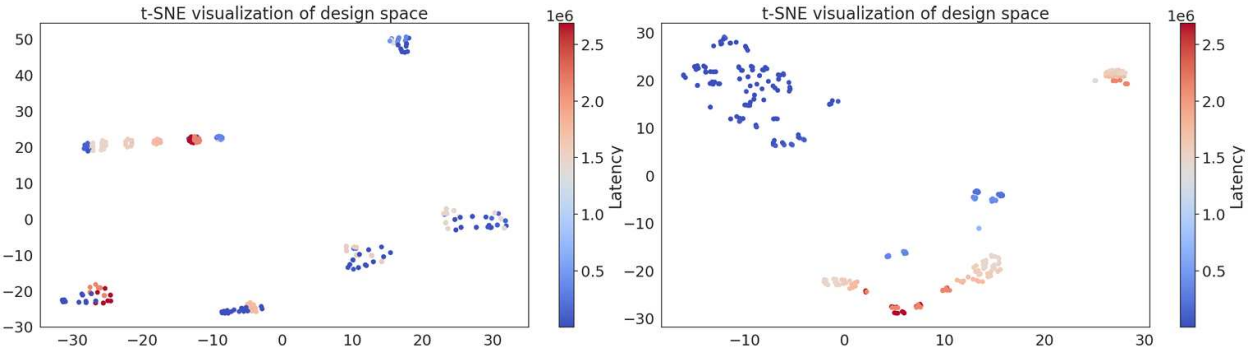



**Figure 3.26:** Visualization of the design configurations of a stencil via t-SNE. Each point represents a design with colors indicating its latency value. Left: Designs represented by the initial features. Right: Designs represented by the embeddings learned by GNN-DSE. Taken from [73].

**"HARP," or how GNN-DSE can be improved by adding pseudo-nodes and decoupling the representation of programs and its transformations.**

A common issue in GNNs is their performance degradation as the number of layers increases, leading to a phenomenon known as over-smoothing [70]. This occurs when repeated graph convolutional layers create too similar node embeddings, thus losing important info about the graph structure. This work addresses the long-range dependency issue in HLS kernels by proposing a hierarchical graph structure, reducing the average shortest path in our benchmark kernels by 5×.

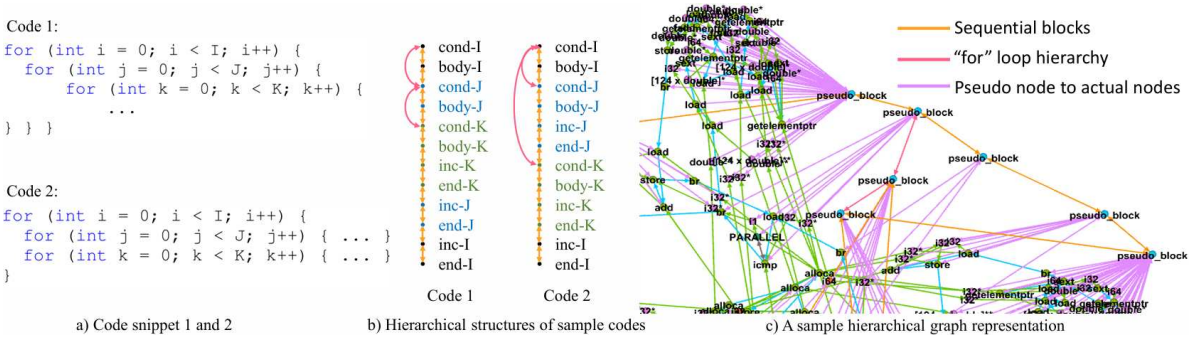



**Figure 3.27:** Example of how pseudo nodes shorten paths between actual nodes. Taken from [70].

In addition to that, the program structure and its transformations (in the form of pragmas) are decoupled to enhance the model's performance. This introduces a neural pragma transformer (NPT), which models pragmas as learnable functions applied to the program representation.

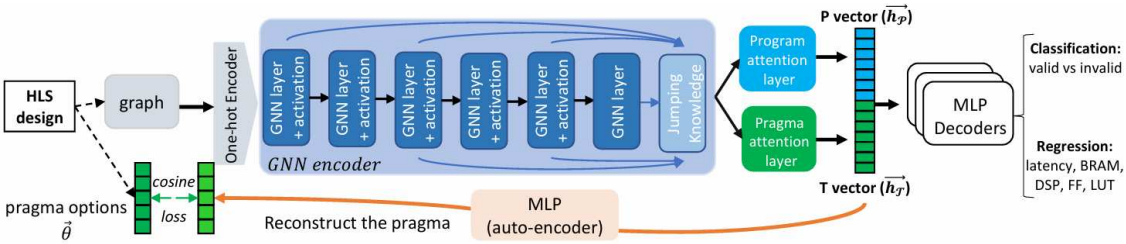



**Figure 3.28:** High-level overview of the HARP framework. Taken from [70].

Each point in Figure 3.29 represents a different design point from a different kernel and is color-coded based on its kernel name. The GNN-DSE embeddings are interleaved when labeled by kernel name, whereas HARP successfully clusters them based on the kernel they belong to [70].

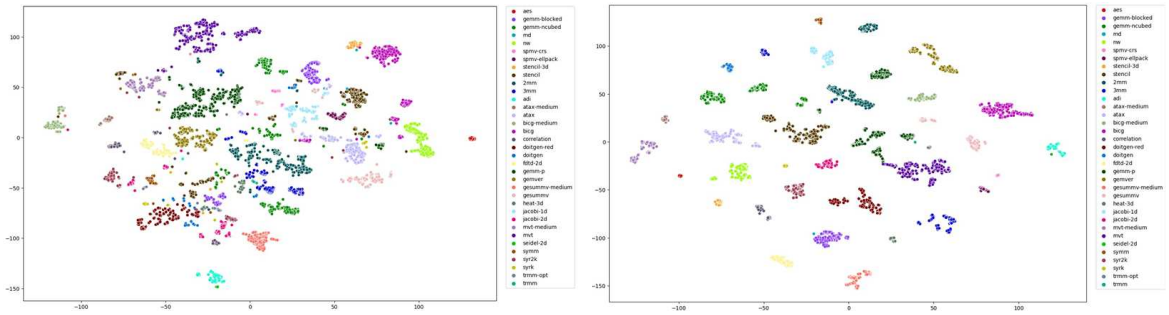


**Figure 3.29:** Comparison between GNN-DSE (left) and HARP (right) t-SNE visualizations of generated graph embeddings, color-coded by kernel name. Taken from [70].

**"Python_graphs," an open source library for representing Python programs as graphs, or how to augment ASTs with control-flow data for its use in machine learning.**

The most common graph representations of source code are the abstract syntax tree (AST), control-flow graph (CFG), data-flow graphs, inter-procedural control-flow graph (ICFG), interval graph, and composite "program graphs" that encode information from multiple of the aforementioned graphs, possibly with additional program-derived data [75].

Python_graphs library's functionalities are generating control-flow graphs, performing data-flow analyses, generating composite program graphs, and computing the cyclomatic complexity of Python programs. The python_graphs library implements a single kind of composite program graph based on [76]. A program graph has the AST of the program it represents as its backbone. Each node in the program graph directly corresponds to a single node in the AST, and vice versa.

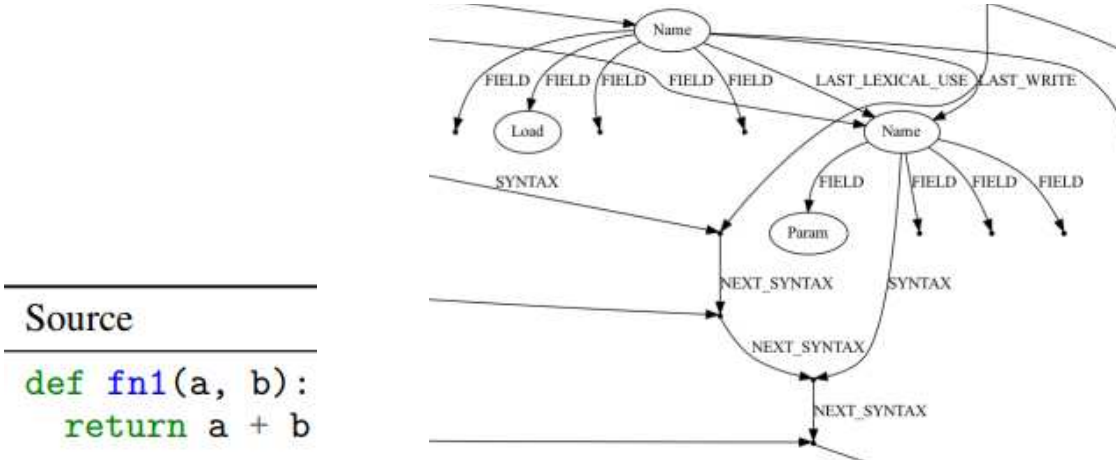



**Figure 3.30:** Example program and a tiny portion of its associated program graph. Adapted from [75].

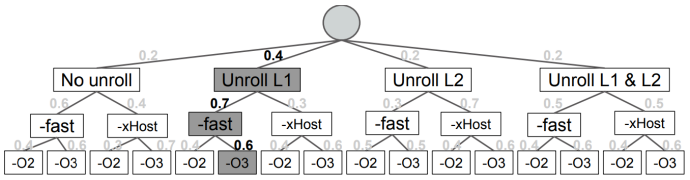
According to [75], each syntax element in the program (leaf nodes in the AST) corresponds to a syntax node in the program graph. Each edge in the AST also appears in the program graph. The program graph then has additional edges representing the following relationships between program pieces: next-syntax, last-lexical-use, cfg-next, last-read, last-write, computer-from, calls, formal-arg-name, and returns-to. Collectively, the edges in a program graph convey control-flow, data-flow, lexical, and syntactic information about the program.

**Extending to efficient search in big optimization spaces for non-FPGA kernels scheduling.**

A kernel is defined as an abstract concept that supports different concrete implementations, each called "kernel schedule". Kernel schedules differ due to the application of code transformations, such as tiling, thread blocking, and loop unrolling, merging, or splitting [77]. Kernel scheduling is an optimization problem whose objective function is running time: the faster a kernel runs, the better that schedule is. It involves experimenting with compiler directives, environment variables, and compiler flags; all these influence performance, but their combinations are unpredictable [78]. The problem of finding exact analytical solutions to kernel scheduling is open. Typical techniques are stochastic, relatively slow, and provide no guarantees of optimality [77]. Usually, a search technique is utilized in a vast space of feasible performance optimization techniques. There are two properties of the search space that make finding the optimal set of optimization techniques difficult: its size and the non-smoothness of the impact on performance [78].

**Performance Optimization Search Engine enhanced by User's Prior Knowledge**

The authors propose Open-Case, a Search-Space Description Language (SSDL), and implemented an engine that tests the performance impact of optimization options by executing optimized programs and checking their results. Two search algorithms are used: random and case-tree.




```
[Environmental variables]
    OMP_NUM_THREADS=16
[Compiler flags]
    -openmp -O3 -align array64byte -opt-prefetch=0
[Compiler directives]
    !DEC$ ATTRIBUTES ALIGN, !DEC$ ASSUME_ALIGNED
    !DEC$ vector always aligned
    !$OMP END DO nowait, and !DEC\$ noinline
[Source transformations]
    Loop unroll, loop merge, array-merge, pre-calculation
```

**Figure 3.31:** A tree representation of the search space generated from all performance optimization techniques considered (left) and an example of one optimization (right). Taken from [78].

OpenCase achieves better performance than conventional compilers because it can generate search spaces based on the user's prior knowledge of performance optimization. It achieved around 1.5x speed-ups from the original performance, but there is still room for improvement, as it only achieved around 50% performance compared to the results from manual optimization.

# Chapter 4

# Conclusion

The common thread of this work is to create the foundation for e-graph-based approaches and tools for optimizing programs, such as auto-parallelization. We covered a manual parallelization case study and are working on the first part of auto-parallelizing program synthesis.

We have demonstrated the potential of equality saturation-based code-to-graph representations in performance characterization, an important aspect of time-critical systems optimization.

Our e-graph approach stands out from other code-to-graph methods due to its unique features. First, our graph representation is "compositional", meaning that similar code regions correspond to similar structures within the e-graph. This feature enhances the e-graph's ability to capture patterns in the code. Second, our approach leverages the synergy between the more compact representation and Graph ML algorithms. This means that less effort is required to map embedding vectors to performance features, such as the number of loops, leading to more efficient performance characterization. Third, our e-graph approach, Egg-no-graph, is not limited to capturing control flow information like CFGs and SSAs. It also excels in identifying data dependencies in the program. For instance, different instructions or expressions that utilize the same variable are stored only a few hops away in our e-graph, even if they are many basic blocks away in the corresponding SSA. This comprehensive data representation is a key strength of our approach. Fourth, in addition to all the above, the Egg-no-graph is also more fine-grained than SSA because the reordering of statements leads to different graph structures in the e-graph. Fifth, the e-graph-based approach is supposed to capture dependencies between program expressions. It aims to approximate a fast & lightweight static analysis; this work can be extended by formally analyzing our e-graphs.

Our results not only motivate further research in Graph ML-based performance characterization but also open up a promising new horizon in heterogeneity-aware automatic optimization of time-critical systems. We aim to guide our future research efforts to verifiable approaches and dive into more real-world study cases, including but not limited to natural threats impact mitigation.

# Bibliography


[1] W. Hanka, J. Saul, B. Weber, J. Becker, P. Harjadi, Fauzi, and GITEWS Seismology Group. Real-time earthquake monitoring for tsunami warning in the indian ocean and beyond. *Natural Hazards and Earth System Sciences*, 10(12):2611–2622, 2010.

[2] Pankaj Kumar, Kamal, M. Sharma, Pratibha, Ravi Jakka, Ashok Kumar, G. Joshi, and Piyoosh Rautela. Successful alert issuance with sufficient lead time by uttarakhand state earthquake early warning system: Case study of nepal earthquakes. *Journal of the Geological Society of India*, 99:303–310, 03 2023.

[3] Ocione D. Filho, Tiffany Mendonça, Gabriel Dietzsch, Bruno Miranda, Felipe O. Passos, Thales Sehn Körting, Gilberto Ribeiro de Queiroz, Luciano Pezzi, Douglas F. M. Gherardi, and Laércio M. Namikawa. Assessment of the impacts of the 2023 earthquake in diyarbakir, turkey with CBERS-4A satellite images. In *XXIV Brazilian Symposium on Geoinformatics - GEOINFO, São José dos Campos, Brazil, December*, pages 167–174. MCTI/INPE, 2023.

[4] Jin Koo Lee, Jeongbeom Seo, and Sung Whang. A study on the design of ground motion database and processing for input seismic evaluation and nuclear power plant safety. *Transactions of the Korean Nuclear Society Spring Meeting*, 05 2023.

[5] Ashok Kumar, Himanshu Mittal, Rajiv Sachdeva, and Arjun Kumar. Indian strong motion instrumentation network. *Seismological Research Letters*, 83:59–66, 01 2012.

[6] David Boore and Julian Bommer. Processing of strong-motion accelerograms: Needs, options and consequences. *Soil Dynamics and Earthquake Engineering*, 25:93–115, 02 2005.

[7] Denise Cervelli, P. Cervelli, and T. Murray. New software for long-term storage and analysis of seismic wave data. *AGU Fall Meeting Abstracts*, -1:0705, 11 2004.

[8] A. Caserta, V. Ruggiero, and P. Lanucara. Numerical modelling of dynamical interaction between seismic radiation and near-surface geological structures: a parallel approach. *Computers & Geosciences*, 28(9):1069–1077, 2002.

[9] Kinemetrics Newsletter. Kinemetrics executes 3rd contract with caltech for california's earthquake early warning system, Mar 2017.

[10] Axel Lloret. Basalt accelerograph cuyo digital photo repository, argentina, Sep 2020.

[11] Jedidiah McClurg, Miles Claver, Jackson Garner, Jake Vossen, Jordan Schmerge, and Mehmet E. Belviranli. Optimizing regular expressions via rewrite-guided synthesis. In *PACT*, pages 426–438. ACM, 2022.

[12] Ismet Dagli, Alexander Cieslewicz, Jedidiah McClurg, and Mehmet E. Belviranli. Axonn: energy-aware execution of neural network inference on multi-accelerator heterogeneous socs. In *DAC*, pages 1069–1074. ACM, 2022.

[13] Daniel Mawhirter, Sam Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu. Dryadic: Flexible and fast graph pattern matching at scale. In *PACT*, pages 289–303. IEEE, 2021.

[14] Marco Massa, Francesca Pacor, Lucia Luzi, Dino Bindi, Giuliano Milana, Fabio Sabetta, Antonella Gorini, and Sandro Marcucci. The italian accelerometric archive (itaca): processing of strong-motion data. *Bulletin of Earthquake Engineering*, 8(5):1175–1187, 10 2010.

[15] Dennis Lemus. Monthly seismic activity bulletin, Dec 2023.

[16] Michail Ntinalexis, Pauline P Kruiver, Julian J Bommer, Elmer Ruigrok, Adrian Rodriguez-Marek, Ben Edwards, Rui Pinho, Jesper Spetzler, Edwin Obando Hernandez, Manos Pefkos, Mahdi Bahrampouri, Erik P van Onselen, Bernard Dost, and Jan van Elk. A database of ground motion recordings, site profiles, and amplification factors from the groningen gas field in the netherlands. *Earthquake Spectra*, 39(1):687–701, 2023.

[17] Rasoul Afsari, Saman Nadizadeh Shorabeh, Amir Reza Bakhshi Lomer, Mehdi Homaee, and Jamal Jokar Arsanjani. Using artificial neural networks to assess earthquake vulnerability in urban blocks of tehran. *Remote. Sens.*, 15(5):1248, 2023.

[18] A Zsarnóczay and GG Deierlein. Pelicun: A computational framework for estimating damage, loss and community resilience. In *Proceedings, 17th World Conference on Earthquake Engineering, (Sendai: WCEE)*, 2020.

[19] Lion Krischer, Tobias Megies, Robert Barsch, Moritz Beyreuther, Thomas Lecocq, Corentin Caudron, and Joachim Wassermann. Obspy: A bridge for seismology into the scientific python ecosystem. *Computational Science & Discovery*, 8:014003, 05 2015.

[20] Derrick J. A. Chambers, M. Shawn Boltz, and Calum J. Chamberlain. Obsplus: A pandas-centric obspy expansion pack. *J. Open Source Softw.*, 6(60):2696, 2021.

[21] Jan Baczek, Dmytro Zhylko, Gilberto Titericz, Sajad Darabi, Jean-Francois Puget, Izzy Putterman, Dawid Majchrowski, Anmol Gupta, Kyle Kranen, and Pawel Morkisz. TSPP: A unified benchmarking tool for time-series forecasting. *CoRR*, abs/2312.17100, 2023.

[22] Guillermo Corneio-Surez, Leonardo Van der Laat, Esteban Meneses, Javier Pacheco, and Mau Mora. Using parallel computing for seismo-volcanic event location based on seismic amplitudes. In *IEEE 38th Central America and Panama Convention (CONCAPAN)*, 2018.

[23] Marco Olivieri and John Clinton. An almost fair comparison between earthworm and seiscomp3. *Seismological Research Letters*, 83(4):720–727, 2012.

[24] Celso Reyes and Michael West. The waveform suite: A robust platform for manipulating waveforms in matlab. *Seismological Research Letters*, 82:104–110, 01 2011.

[25] Marco Carratù, Vincenzo Gallo, Vincenzo Paciello, and Antonio Pietrosanto. A deep learning approach for the development of an early earthquake warning system. In *IEEE International Instrumentation and Measurement Technology Conference, I2MTC 2022, Ottawa, ON, Canada, May 16-19, 2022*, pages 1–6. IEEE, 2022.

[26] Bhanu Chamoli, Ashok Kumar, Ajay Gairola, Ravi Jakka, Himanshu Mittal, and Amit Srivastava. Development of earthquake early warning system. In *6th Annual Conference of the International Society for Integrated Disaster Risk Management*, 10 2015.

[27] Govind Rathore, Pankaj Kumar, Mukat Sharma, Kamal, Ravi Jakka, and Ashok Kumar. *Development and Implementation of a Regional Earthquake Early Warning System in Northern India*, pages 537–544. 07 2023.

[28] Ljupco Jordanovski, Boro Jakimovski, and Anastas Misev. Massively parallel seismic data wavelet processing using advanced grid workflows. In Danco Davcev and Jorge Marx Gómez, editors, *ICT Innovations, Ohrid, Macedonia, September*, pages 411–419. Springer, 2009.

[29] Peter Goldstein and A Snoke. Sac availability for the iris community. *Incorporated Institutions for Seismology Data Management Center Electronic Newsletter*, 7, 01 2005.

[30] Brian Savage. sacio: A library for seismic analysis code data files. *Open Source Softw.*, 2021.

[31] R. Puglia, E. Russo, L. Luzi, M. D'Amico, C. Felicetta, F. Pacor, and G. Lanzano. Strong-motion processing service: a tool to access and analyse earthquakes strong-motion waveforms. *Bulletin of Earthquake Engineering*, 16(7):2641–2651, 07 2018.

[32] Jens Havskov, Peter H. Voss, and Lars Ottemöller. Seismological observatory software: 30 yr of seisan. *Seismological Research Letters*, 91(3):1846–1852, 03 2020.

[33] Sepideh J. Rastin, Charles P. Unsworth, Ken R. Gledhill, George G. Coghill, Mark Chadwick, and Russell Robinson. PQLX noise model for the URZ station in the matata region of new zealand. In *10th International Conference on Information Sciences, Signal Processing and their Applications, ISSPA 2010, Kuala Lumpur, Malaysia, 2010*, pages 17–20. IEEE, 2010.

[34] Arjun Kumar, Ashwani Kumar, Ashok Kumar, Himanshu Mittal, and Rakhi Bhardwaj. Software to estimate spectral and source parameters. *International Journal of Geosciences*, 2012.

[35] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications, 2018.

[36] A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[37] Shima Khoshraftar and Aijun An. A survey on graph representation learning methods, 2022.

[38] Van Thuy Hoang, Hyeon-Ju Jeon, Eun-Soon You, Yoewon Yoon, Sungyeop Jung, and O-Joun Lee. Graph representation learning and its applications: A survey. *Sensors*, 23(8), 2023.

[39] Fenxiao Chen, Yun-Cheng Wang, Bin Wang, and C.-C. Jay Kuo. Graph representation learning: a survey. *APSIPA Transactions on Signal and Information Processing*, 9(1):–, 2020.

[40] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008.

[41] William L. Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159.

[42] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '14. ACM, August 2014.

[43] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks, 2016.

[44] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[45] Nils Kriege. *Comparing Graphs: Algorithms & Applications*. PhD thesis, 01 2015.

[46] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten Borgwardt. Efficient graphlet kernels for large graph comparison. In *Proceedings of the 12th International Conference on AI and Statistics*, pages 488–495. PMLR, 2009.

[47] Risi Kondor, Nino Shervashidze, and Karsten M. Borgwardt. The graphlet spectrum. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 529–536, New York, NY, USA, 2009. Association for Computing Machinery.

[48] Nino Shervashidze and Karsten Borgwardt. Fast subtree kernels on graphs. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc., 2009.

[49] Adrien Leman. The reduction of a graph to canonical form and the algebra which appears therein. 2018.

[50] Christopher Morris, Kristian Kersting, and Petra Mutzel. Glocalized weisfeiler-lehman graph kernels: Global-local feature maps of graphs. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 327–336, 2017.

[51] Matteo Togninalli, Elisabetta Ghisu, Felipe Llinares-López, Bastian Rieck, and Karsten Borgwardt. Wasserstein weisfeiler-lehman graph kernels, 2019.

[52] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(77):2539–2561, 2011.

[53] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, page 1067–1077, 2015.

[54] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[55] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph cnns, 2017.

[56] Ruoyu Li and Junzhou Huang. Learning graph while training: An evolving graph convolutional neural network, 2017.

[57] Hoang NT and Takanori Maehara. Revisiting gnns: All we have is low-pass filters, 2019.

[58] Fei Jiang, Lifang He, Yi Zheng, Enqiang Zhu, Jin Xu, and Philip S. Yu. On spectral graph embedding: A non-backtracking perspective and graph approximation, 2018.

[59] Bijaya Adhikari, Yao Zhang, Naren Ramakrishnan, and B. Aditya Prakash. Sub2vec: Feature learning for subgraphs. In *Advances in Knowledge Discovery and Data Mining*, pages 170–182, Cham, 2018. Springer International Publishing.

[60] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, January 2021.

[61] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, March 2011.

[62] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, and Zachary Tatlock. Better together: Unifying datalog and equality saturation, 2023.

[63] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. volume 9780, pages 422–430, 07 2016.

[64] Tomofumi Yuki. Understanding polybench/c 3.2 kernels. In *International workshop on polyhedral compilation techniques (IMPACT)*, pages 1–5, 2014.

[65] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[66] E Bendersky. Github–eliben/pycparser: Complete c99 parser in pure python, 2019.

[67] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. Robust gnn-based representation learning for hls. In *In Proceedings of the 42nd IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023.

[68] Ali TehraniJamsaz, Quazi Ishtiaque Mahmud, Le Chen, Nesreen K. Ahmed, and Ali Jannesari. Perfograph: A numerical aware program graph representation for performance optimization and program analysis, 2023.

[69] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. Programl: Graph-based deep learning for program optimization and analysis, 2020.

[70] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. Robust gnn-based representation learning for hls. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.

[71] Mingyang Kou, Jun Zeng, Boxiao Han, Fei Xu, Jiangyuan Gu, and Hailong Yao. Geml: Gnn-based efficient mapping method for large loop applications on cgra. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, page 337–342, New York, NY, USA, 2022. Association for Computing Machinery.

[72] Zizheng Guo, Mingjie Liu, Jiaqi Gu, Shuhan Zhang, David Z. Pan, and Yibo Lin. A timing engine inspired graph neural network model for pre-routing slack prediction. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, page 1207–1212, New York, NY, USA, 2022. Association for Computing Machinery.

[73] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. Enabling automated fpga accelerator optimization using graph neural networks, 2021.

[74] Benjamin Carrion Schafer and Zi Wang. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39:2628–2639, 2020.

[75] David Bieber, Kensen Shi, Petros Maniatis, Charles Sutton, Vincent Hellendoorn, Daniel Johnson, and Daniel Tarlow. A library for representing python programs as graphs for machine learning, 2022.

[76] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[77] Michael Canesche, Vanderson Rosário, Edson Borin, and Fernando Quintão Pereira. The droplet search algorithm for kernel scheduling. *ACM Trans. Archit. Code Optim.*, May 2024.

[78] Youngsung Kim, Pavol Černý, and John Dennis. Performance search engine driven by prior knowledge of optimization. In *2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'15, NY, USA, 2015.

# Appendix A

# Egg-no-graph: Design details

There are 24 ways of linking e-nodes in eggnog, which Egglog calls "functions"; please note the use of lowercase to show the difference with our class type, "Function", which stands for actual Functions and Procedures (functions that do not have a return statement).

```
@egraph.class_
class BasicBlock(Expr)
class Var(Expr)
class Op(Expr)
class Function(Expr)

@egraph.function
def PHI(l: Var, c: Var, r: Var) -> Op
def Gt(l: Var, r: Var) -> Op
def Gte(l: Var, r: Var) -> Op
def Lt(l: Var, r: Var) -> Op
def Lte(l: Var, r: Var) -> Op
def Eq(l: Var, r: Var) -> Op
def Neq(l: Var, r: Var) -> Op
```

```
def Mul(l: Var, c: Var, r: Var) -> Op
def Add(l: Var, c: Var, r: Var) -> Op
def Sub(l: Var, c: Var, r: Var) -> Op
def Div(l: Var, c: Var, r: Var) -> Op
def Mod(l: Var, c: Var, r: Var) -> Op
def Seq(l: Op, r: Op) -> Op
def GoTo(c: BasicBlock) -> Op
def If(l: Op, c: Op, r: Op) -> Op
def Start(c: Op) -> BasicBlock
def Assign(l: Var, r: Var) -> Op
def Array(l: Var, r: Var) -> Var
def Matrix(l: Vr, c: Vr, r: Vr) -> Vr
def Caller(c: Function) -> Var
def Procedure(c: Function) -> Op
def Entry(c: BasicBlock) -> Function
def Exit(c: BasicBlock) -> Function
```

**Listing A.1:** Eggnog's classes and functions.

Egg-no-graph creation has linear complexity over the content of a given SSA graph, meaning that each SSA feature is visited only once, eliminating the need for backtracking. **Algorithm 1**, used to convert an SSA to its Eggnog representation is explained as follows:

- We create the sets BB, VAR, and FUN; which will store all basic blocks, variables, and functions/procedures. Each SSA graph contains one or more functions. It is important to note that functions are not connected between them; this means there is no edge in the graph to link a function's call with its definition (its first basic block).
- There is a main while-loop which will be executed per each SSA instruction, there are four possibilities: functions, entry or exit-basic blocks (first and last BBs of a function), edges (stored by gcc always at the end of a function definition), and all remaining basic blocks; for the latter, its content is shown in **Algorithm 2**, due to space constraints.
- Lastly, we link the function's calls and returns (or simply the end, in the case of procedures) with the function's definition; this means the first and last basic blocks.

**Algorithm 3** SSA2Eggnog algorithm

**Require:** SSA ▷ File stream of SSA.dot content

egraph = EGraph()
BB, VAR, FUN = ∅
**while** next(SSA) ≠ End-of-file **do** ▷ Only one pass is needed
    **if** SSA ∈ $S_{Function}$ **then**
        fun_name ← parse(SSA)
        FUN[fun_name] ← egraph.constant(Function)
    **else if** SSA ∈ $S_{Entry}$ or SSA ∈ $S_{Exit}$ **then** ▷ Entry and Exit basic blocks
        fn, bb ← parse(SSA)
        BB[fn][bb] ← egraph.constant(BasicBlock)
    **else if** SSA ∈ $S_{Edge}$ **then** ▷ All edges
        $fn_{source}$, $bb_{source}$, $fn_{dest}$, $bb_{dest}$ ← parse(SSA)
        egraph.register(union(BB[$fn_{source}$][$bb_{source}$]).with(BB[$fn_{dest}$][$bb_{dest}$]))
    **else if** SSA ∈ $S_{BasicBlock}$ **then** ▷ Remaining basic blocks
        fn, bb ← parse(SSA)
        BB[fn][bb] ← egraph.constant(BasicBlock)
        execute Basic Block Content Processing ▷ Algorithm 2
**for** each e in FUN **do**
    egraph.register(union(e).with(egraph.let( Entry(BB[e[fn]][2]) )))
    egraph.register(union(e).with(egraph.let( Exit(BB[e[fn]][1]) )))
**return** egraph

**Algorithm 2** is in charge of processing the content of all basic blocks, detailed as follows:

- First, declare the "OP" set, which will contain all BB operations (aka instructions).
- If statements require storing all variables, comparison operators ($>, \geq, <, \leq, =, \neq$), goto instruction for both destination BBs, and the if itself (blue).
- Binary operations ($*, +, -, /$,mod) require storing all variables and the operation itself (blue).
- Goto and return instructions, an e-node is created with the linked basic block (blue).
- Assignments, the instruction which has the most possible specific scenarios. Depending on the case, we store function calls, array or matrix indexing, or variable storing. Lastly, an e-node is created, linking both the left and right-hand sides of the instruction (blue).
- Procedure calls are stored (blue) and then linked to its definition.
- We create an "sequence" e-node between each operation inside the OP set (purple). This has to be implemented backward to ensure the original SSA order is maintained.
- Finally, the first instruction of the BB (blue) is linked to the BB itself.

**Algorithm 4** Basic Blocks Content Processing

OP = $\varnothing$, $C_{constants} = 0$
**while** next(SSA) $\neq$ End-of-BB **do**
  **if** SSA $\in S_{phi}$ **then**
    v, a, b, link_name $\leftarrow$ parse(SSA)
    VAR[$x$] $\leftarrow$ egraph.constant($x$, Var) $\forall x \in$ {v, a, b}
    OP = OP $\cup$ { egraph.let( PHI(VAR[v], VAR[a], VAR[b]) ) }
  **else if** SSA $\in S_{if}$ **then**
    lhs, c, rhs, $\text{fn}_\text{a}$, $\text{bb}_\text{a}$, $\text{fn}_\text{b}$, $\text{bb}_\text{b}$ $\leftarrow$ parse(SSA) ▷ c $\in$ {Gt, Gte, Lt, Lte, Eq, Neq}
    $v \leftarrow$ rename($v$, fn, bb, $C_{const}$) $\forall v \in$ {lhs, rhs} ▷ Updates $C_{const}$
    VAR[$x$] $\leftarrow$ egraph.constant($x$, Var) $\forall x \in$ {lhs, rhs}
    comp $\leftarrow$ egraph.let( c(VAR[lhs], VAR[rhs]) )
    $\text{goto}_\text{x} \leftarrow$ egraph.let( GoTo(BB[$\text{fn}_\text{x}$][$\text{bb}_\text{x}$]) ) $\forall x \in$ {a, b}
    OP = OP $\cup$ { egraph.let( If(comp, $\text{goto}_\text{a}$, $\text{goto}_\text{b}$) ) }
  **else if** SSA $\in S_{op}$ **then**
    lhs, op1, op, op2 $\leftarrow$ parse(SSA) ▷ op $\in$ {Mul, Add, Sub, Div, Mod}
    $v \leftarrow$ rename($v$, fn, bb, $C_{const}$) $\forall v \in$ {lhs, op1, op2} ▷ Updates $C_{const}$
    VAR[$x$] $\leftarrow$ egraph.constant($x$, Var) $\forall x \in$ {lhs, op1, op2}
    OP = OP $\cup$ { egraph.let( op(VAR[lhs], VAR[op1], VAR[op2]) ) }
  **else if** SSA $\in S_{goto}$ or SSA $\in S_{return}$ **then**
    fn, bb $\leftarrow$ parse(SSA)
    OP = OP $\cup$ { egraph.let( GoTo(BB[fn][bb]) ) } ▷ When return happen: $\text{bb}_\text{return}$=1
  **else if** SSA $\in S_{assignment}$ **then**
    lhs{var,arr,mat}, rhs{var,arr,mat,fun} $\leftarrow$ parse(SSA)
    $v \leftarrow$ rename($v$, fn, bb, $C_{const}$) $\forall v \in$ {lhs, rhs} ▷ Updates $C_{const}$
    VAR[$x$] $\leftarrow$ egraph.constant($x$, Var) $\forall x \in$ {lhs, rhs}
    **if** $\text{rhs}_\text{fun} \neq \varnothing$ **then** ▷ Repeat for $\text{lhs}_\text{final}$
      $\text{rhs}_\text{final} \leftarrow$ egraph.let( Caller(FUN[$\text{rhs}_\text{fun}$]) )
    **else if** $\text{rhs}_\text{mat} \neq \varnothing$ **then**
      $\text{rhs}_\text{final} \leftarrow$ egraph.let( Matrix(VAR[$\text{rhs}_\text{var}$],VAR[$\text{rhs}_\text{arr}$],VAR[$\text{rhs}_\text{mat}$]) )
    **else if** $\text{rhs}_\text{arr} \neq \varnothing$ **then**
      $\text{rhs}_\text{final} \leftarrow$ egraph.let( Array(VAR[$\text{rhs}_\text{var}$],VAR[$\text{rhs}_\text{arr}$]) )
    **else** $\text{rhs}_\text{final} \leftarrow$ VAR[$\text{rhs}_\text{var}$]
    OP = OP $\cup$ { egraph.let( Assign($\text{lhs}_\text{final}$,$\text{rhs}_\text{final}$) ) }
  **else if** SSA $\in S_{procedure}$ **then**
    fun_name $\leftarrow$ parse(SSA)
    OP = OP $\cup$ { egraph.let( Procedure(FUN[fun_name]) ) }
**if** |OP| = 1 **then** ▷ If there is only one, there is no need for seq
  head $\leftarrow \text{OP}_0$
**else**
  head $\leftarrow$ egraph.let( Seq($\text{OP}_{n-2}$, $\text{OP}_{n-1}$) )
  **for** i $\leftarrow$ [n-2, 1] step $-1$ **do** ▷ Merge all operations with seq
    head $\leftarrow$ egraph.let( Seq($\text{OP}_{i-1}$, head) )
egraph.register(union(BBs[fn][bb]).with(egraph.let( Start(head) )))