# CommAnalyzer: Automated Estimation of Communication Cost on HPC Clusters Using Sequential Code

Ahmed E. Helal, Changhee Jung, Wu-chun Feng, Yasser Y. Hanafy
Electrical & Computer Eng. and Computer Science, Virginia Tech
ammhelal,chjung,wfeng,yhanafy@vt.edu

## ABSTRACT

MPI+X is the de facto standard for programming applications on HPC clusters. The performance and scalability on such systems is limited by the communication cost on different number of processes and compute nodes. Therefore, the current communication analysis tools play a critical role in the design and development of HPC applications. However, these tools require the availability of the MPI implementation, which might not exist in the early stage of the development process due to the parallel programming effort and time. This paper presents CommAnalyzer, an automated tool for communication model generation from a sequential code. CommAnalyzer uses novel compiler analysis techniques and graph algorithms to capture the inherent communication characteristics of sequential applications, and to estimate their communication cost on HPC systems. The experiments with real-world, regular and irregular scientific applications demonstrate the utility of CommAnalyzer in estimating the communication cost on HPC clusters with more than 95% accuracy on average.

## KEYWORDS

Distributed Computing, Communication Analysis, Performance Modeling, Prediction, Parallel Architectures, HPC, MPI, LLVM

## 1 INTRODUCTION

In order to scale to a large number of compute units, HPC cluster architectures adopt the distributed memory model. These architectures are more difficult to program than shared-memory models and require explicit decomposition and distribution of the program data and computations, due to the lack of a single global address space. The MPI programming model is the de facto standard for programming applications on HPC clusters [6, 14]. MPI uses explicit messaging to exchange data across processes that reside in separate address spaces, and it is often combined with shared-memory programming models, such as OpenMP [35] and OpenACC [34], to exploit the available compute resources seamlessly both within a node and across nodes. On the other hand, the partitioned global address space (PGAS) programming models (e.g., Chapel [13] and UPC [10]) abstract explicit communication and data transfers using distributed memory objects; however, the user still needs to manage the data distribution and data locality across compute nodes.

The application performance and scaling on HPC clusters are limited by the communication cost on different number of compute nodes. In particular, the asymptotic scalability/efficiency of a program on distributed-memory architectures is determined by the growth of the communication as a function of the problem size and the number of processes/nodes [15, 17]. Therefore, researchers have created scalability analysis tools [5, 9, 11, 16, 49, 50] and communication pattern detection tools [4, 27, 38] to study the effect of the data communication on the application performance and to provide valuable insights on the optimization of the communication bottlenecks. While these tools play a critical role in the design and development of HPC applications, they require the availability of the MPI parallel implementation of the original application to construct the communication model. across nodes. Unfortunately, such a communication model depends on the specific MPI implementation rather than the inherent communication characteristics of the original application. Moreover, due to the parallel programming effort and time, the MPI implementation is often not available in the early stages of the development process. Thus, there is a compelling need to analyze the sequential applications and predict the communication cost of their parallel execution on HPC clusters for estimating the scalability and the performance beforehand.

To this end, this paper presents CommAnalyzer, an automated tool for communication model generation from a sequential code, to figure out the scaling characteristics of running the code in parallel on a distributed parallel system using the single program multiple data (SPMD) execution model. The key idea is to reformulate the problem of estimating the communication of parallel program between HPC nodes into that of analyzing the inherent communication properties of the sequential application. To characterize such inherent data communication properties, CommAnalyzer uses novel compiler-based analyses that build *value communication graph* (VCG) from a sequential code taking into account the data-flow behaviors of its program values.

To assess the scalability of the parallel execution, CommAnalyzer in turn leverages graph partitioning algorithms over the VCG to automatically identify sender/receiver entities and to estimate their communication cost on different numbers of HPC cluster nodes. Since parallel programs are typically optimized to minimize the communication between compute nodes, their communication volume is likely to serve as a cut for partitioning the VCG. Similarly, CommAnalyzer treats the group of VCG vertices that are tightly-coupled as the communication entities, i.e., sender and receiver. Finally, CommAnalyzer generates the communication matrix across multiple nodes and estimate the total communication cost. As a result, CommAnalyzer can help the users to project the performance of the effective SPMD parallel implementation of their sequential applications in the early stages of the development process, regardless of the target programming model (e.g., MPI or PGAS).

The following are the contributions of this work:

- A novel and automated approach for projecting the communication cost of sequential applications on HPC clusters based on value-flow analysis, value liveness analysis, dynamic program slicing, and graph algorithms. This approach quantifies the inherent communication of sequential applications regardless of

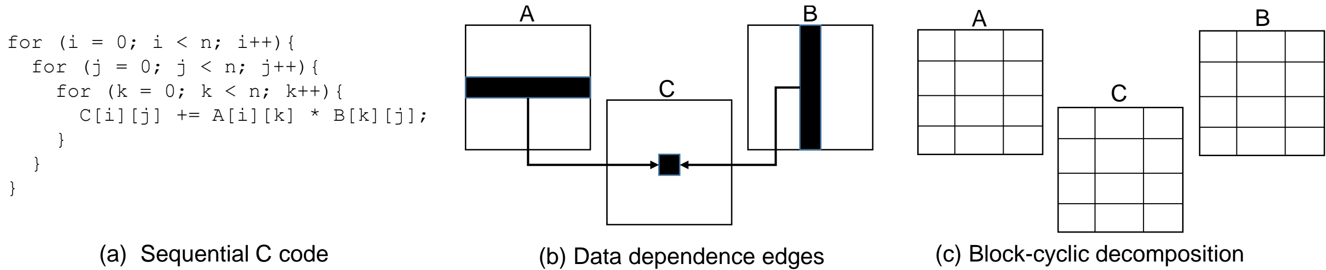



**Figure 1: Simple communication cost estimation for matrix multiplication**

the underlying data structures, and it is applicable for regular, irregular, and unstructured problems (Section 4).

- Model validation using two canonical regular and irregular workloads: matrix multiplication and sparse matrix vector multiplication (Section 5).
- Case studies on three real-world, regular and irregular scientific applications: MiniGhost [7], Heat2D [36] and LULESH[28]. The experiments demonstrate the utility of CommAnalyzer in identifying the communication cost on HPC clusters with more than 95% accuracy on average (Section 5).

# 2 BACKGROUND AND CHALLENGES

## 2.1 Distributed-Memory Execution Model

We assume that the applications running on the target HPC clusters follow the single program multiple data (SPMD) execution model, which is the dominant execution model on such architectures [6, 14]. Figure 2 shows the SPMD execution model, where the program data is partitioned and mapped to the different processes (compute nodes) and all processes execute the same program and perform computations on the data items that they own (i.e., owner-computes rule). The data items owned by each process are stored on its private (local) address space, and when the local computations on a process involve non-local data items, these items are accessed using inter-process communication.

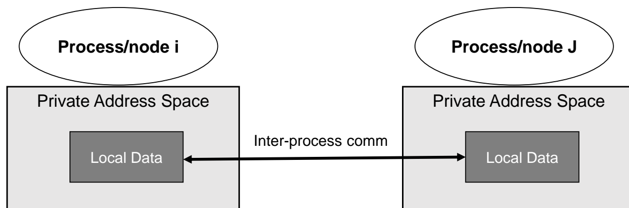



**Figure 2: The SPMD execution model**

## 2.2 Simple Communication Cost Estimation

In the SPMD execution model, the inter-process communication results from dependencies between local and non-local data items. Therefore, a simple approach for estimating the communication cost is to partition the program data and then identify the data dependencies across the different partitions.

Figure 1 shows a simple communication cost estimation for matrix multiplication on distributed-memory architectures. The sequential C code (a) is given to the compiler, and traditional data-flow analysis is used to identify the data dependence [32] (b) between the data items. Next, a domain decomposition method such as block-cyclic (c) is used to partition and distribute the input matrices over the compute nodes. Finally, the communication cost is estimated as the number of data dependence edges between the data items that exist in different compute nodes. This approach has been used by the auto-parallelizing compilers for distributed-memory architectures [22, 48] in the early days of high performance computing.

While this simple communication cost estimation is sufficient for regular and structured problems such as matrix multiplication, real-world scientific applications are challenging and require more sophisticated approaches due to the complex and irregular computation and unstructured memory access patterns.

## 2.3 Challenges

*2.3.1 Indirect Memory Access.* Irregular and unstructured problems arise in many important scientific applications such as hydrodynamics and molecular dynamics [21, 25]. These problems are characterized by an indirect memory access pattern via index data structures, which cannot be determined at compile time. Figure 3 shows an example of such an indirect memory access where a traditional data-flow analysis fails to precisely capture the data dependence between the $x$ and $y$ data arrays whose index is a value determined at runtime, which makes it impossible to figure out which part of $x$ ($y$) is defined (used) due to the lack of runtime information.

```
for( i=0 ; i<numElem ; ++i ){
  for( j=0 ; j<numNodes ; ++i ){
    x[nodeA[i][j]] += y[nodeB[i][j]] * z[i];
  }
}
```

**Figure 3: Indirect Memory Access**

*2.3.2 Workspace Data Structures.* In unstructured problems, it is common to gather several data values from the program-level (domain-level) data structures into workspace (temporary) data structures. The actual computations are performed in the workspace, and then the final results are scattered to the program-level data structures. Usually, there are multiple levels of workspace data structures, i.e., the intermediate results in a workspace are used to compute the values of another workspace.

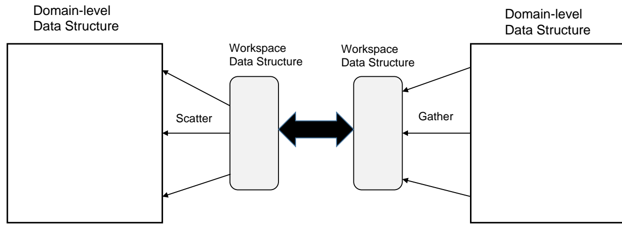



**Figure 4: Workspace data structures mask the true data dependencies**

Figure 4 shows an example of unstructured application that uses such workspace data structures to perform its computations. In this common design pattern, workspace data structures mask the true data dependence edges between the data items of the program-level data structures. Hence, the data-flow analysis ends up generating massive number of data dependence edges between the program-level data items and the workspace data items, which results in inaccurate estimation of the actual communication cost.

*2.3.3 Singleton Data Items.* Most scientific applications have singleton data items, i.e., data items that are used in almost all the computations (e.g., simulation parameters and coefficients) and data items that use the output of theses computations (e.g., simulation error, simulation time step, etc.). Hence, there is a massive number of data dependence edges (communication edges) between these singleton data items and the rest of the program data in the original sequential implementation. However, typical MPI implementations create local copies of the singleton data items in each process and use collective communication messages (e.g., broadcast and allreduce) to update their values at the beginning and the end of the program and/or each time step, instead of accessing their global copy via inter-process communication in every dependent computation. Therefore, the detection of singleton patterns is very important for an accurate estimation of the communication cost.

*2.3.4 Redundant communication.* Computing the communication cost as the number of data dependence edges between data items that exist in different processes is subjected to overestimation. Figure 5 shows an example of the communication overestimation, where two items in process 0 use a single data item in process 1 and its value did not change between the two uses. If there is sufficient memory space at process 0 to store the required data value, it can be read only once from process 1. While the exact data-flow analysis can detect this case in regular applications and remove the redundant communication, it is not possible to do the same for irregular applications due to the indirect memory access.

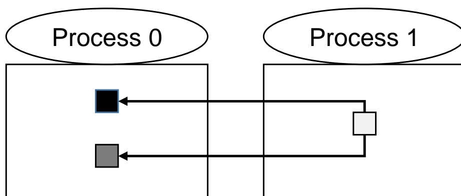



**Figure 5: Redundant Communication**

# 3 OVERVIEW OF COMMANALYZER

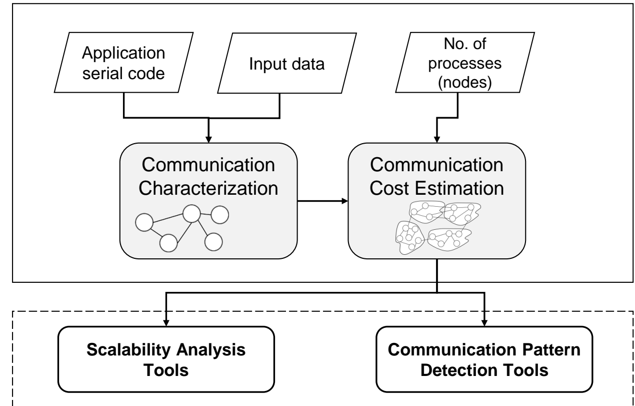



**Figure 6: CommAnalyzer Framework Overview**

Figure 6 shows the proposed framework for the automated generation of communication model from sequential program. The inputs to CommAnalyzer are the sequential application code (in any of the supported languages by the LLVM compiler [31]), representative input data sets, and the number of compute nodes/processes (e.g., [2-64] nodes). The novel dynamic analysis of CommAnalyzer quantifies the inherent communication characteristics of the sequential code and generates the *value communication graph*. This analysis technique supports not only regular problems but also irregular and unstructured problems, that are common in many scientific HPC applications such as hydrodynamics [25]. Next, CommAnalyzer uses graph algorithms to detect the collective broadcast and reduction patterns and to estimate point-to-point communication cost on different number of compute nodes.

CommAnalyzer helps the users and system designers to understand the communication cost of their applications when ported to HPC clusters before investing effort and time in developing the distributed parallel implementation of the original applications. When the projected communication cost of the original sequential code on distributed memory architectures is prohibitive, the users are motivated to consider and develop other algorithms and to explore the different architectural design options for the HPC cluster systems. In addition, by estimating the communication cost from the sequential code rather than the MPI parallel code, CommAnalyzer enables existing communication and scalability analysis tools for MPI applications to serve a wide range users of HPC systems and extend their usability. Such an integrated platform allows the users not only to validate the MPI implementation and optimize the data decomposition and distribution strategy, but also to project the performance of the MPI implementation at large scale before even developing one.

# 4 COMMANALYZER APPROACH

It is a daunting challenge to analyze sequential codes and predict the communication cost of running them in parallel on HPC clusters without the parallel codes. CommAnalyzer relies on the following observation; HPC developers always optimize (minimize) the communication cost of their MPI parallel program across compute nodes. In particular, the communication cost of the resulting

SPMD implementation cannot be smaller than the inherent data-flow (communication) cost of the original sequential program that would otherwise break the program correctness. In other words, analyzing the data communication in the sequential programs can serve as a basis for estimating the communication cost of their parallel implementations that are often highly optimized before running on HPC clusters. However, this presents another challenge, i.e., how to figure out the inherent data communication of the sequential program regardless of its underlying data structures.

The main idea to tackle this challenge is to view the original sequential program as an entity that consumes input values, computes intermediate values and produces output values, as well as to analyze their behaviors. In a sense, these values are small pieces of digital information. Similar to genes (which are small pieces of DNA information), the program values are not constrained by the underlying data structures. Rather, such values can interact, replicate, and flow from one data structure to another, as well as evolve to new values. Actually, the program data structures are mere value containers, i.e., placeholders of the program values. In this view, a single value can exist in more than one memory location, and it can even get killed in its original location (where it was generated) while it remains alive in another location.

To this end, CommAnalyzer adopts a dynamic analysis technique, which is based on dynamic program slicing, to analyze the generation of values, the flow of values, the lifetime of values, and the interactions across values, thereby building *value communication graph* (VCG).

**Algorithm 1** CommAnalyzer Algorithm

**Input: PROGRAM**, **N**
**Output:** $\mathbb{COMM_COST}$
1: $\mathbb{VAL_FC} \leftarrow$ ValFcDetection(**PROGRAM**)
2: $\mathbb{VAL_LIVE} \leftarrow$ ValLiveAnalysis(**PROGRAM**)
3: $\mathbb{VCG} \leftarrow$ CommGraphFormation($\mathbb{VAL_FC}$, $\mathbb{VAL_LIVE}$)
4: $\mathbb{VAL_PMAP} \leftarrow$ ValDecomposition($\mathbb{VCG}$, **N**)
5: $\mathbb{COMM_COST} \leftarrow$ CommEstimation($\mathbb{VAL_FC}$, $\mathbb{VAL_PMAP}$)

Algorithm 1 shows the top-level CommAnalyzer approach which takes the sequential program (along with representative input data) and the number (or range) of compute nodes, and computes the communication cost across these nodes when the program is ported to distributed-memory architectures. In the first place, CommAnalyzer characterizes the inherent communication of the sequential program by detecting the dynamic flow of the program values which is encoded as *value-flow chains* defined as follows:

*Definition 4.1 (Value-Flow Chain (VFC)).* For a given program value, $v$, its value-flow chain consists of $v$ itself and all the other values on which $v$ has data dependence, where the values are defined as a set of unique data observed in memory during program execution.

CommAnalyzer also analyzes the live range [32] (interval) of the program values. Together with VFCs, it is used to generate the *value communication graph* (VCG) of the program. Then, CommAnalyzer decomposes the VCG into multiple partitions to map the program values that are tightly-connected, due to high flow traffic between them, to the same compute node/process. Once the program values are mapped to the different compute nodes, CommAnalyzer uses the owner-computes rule to estimate the communication cost by analyzing the *value-flow chains* across the compute nodes.

## 4.1 Communication Characterization

This section first defines the terminologies used in analyzing the input sequential program and then describes how the inherent communication is understood. CommAnalyzer defines *program* values as a set of unique values observed at runtime, and they are classified into three parts, i.e., *input, output, intermediate* values. The *input* values are the program arguments and any memory location read for the first time at runtime, while the *output* values are the returned data or those that are written to the memory and last until the program termination. During program execution, the values existing in memory locations can be killed with their updates, i.e., losing the original value. That way program values can end up existing in registers and/or memory locations for a limited interval and CommAnalyzer considers them as *intermediate* values. Note, if a value remains in at least one memory location, it is still live thus not a *intermediate* value.

**Algorithm 2** Value-Flow Chain Detection Algorithm

**Input: PROGRAM**
**Output:** $\mathbb{VAL_FC}$
1: *Values* = {} ▷ program values
2: *MemVal* = {} ▷ shadow memory-to-value map
3: **for each:** dynamic store (write) operation $w$ **do**
4: $\quad$ $DS \leftarrow$ DynamicSlicing($w$, **PROGRAM**)
5: $\quad$ $v, s \leftarrow$ ValueFlowAnalysis($DS$, $MemVal$)
6: $\quad$ **if** $v \notin Values$ **then** ▷ writing new value
7: $\quad\quad$ $Values = Values \cup v$
8: $\quad\quad$ $\mathbb{VAL_FC} = \mathbb{VAL_FC} \cup (v\ ,\ s)$
9: $\quad$ **end if**
10: $\quad$ $MemVal$[address($w$)] = $v$
11: **end for**

Algorithm 2 shows the high-level algorithm for calculating the *value-flow chains* (i.e., the inherent communication). Basically, CommAnalyzer needs to identify the unique values observed at runtime and then to investigate the flow across the values by figuring out the dependence in between. It is therefore important to know what value is currently stored in a given memory address during program execution. For this purpose, CommAnalyzer uses a shadow memory (*MemVal*) to keep track of program values that exist in the memory at the granularity of the memory word (e.g., 64 bits for double-precision floating-point data). Thus, each shadow memory location tracks the latest value stored in the corresponding location in the original memory space. Since CommAnalyzer works on the static single assignment (SSA) form [32] of the sequential code (e.g., the LLVM IR), there is no need to track (name) the intermediate values that exist in the registers.

For each store (write) instruction during program execution, CommAnalyzer generates a dynamic program slice from the execution history using the Reduced Dynamic Dependence (RDD) graph

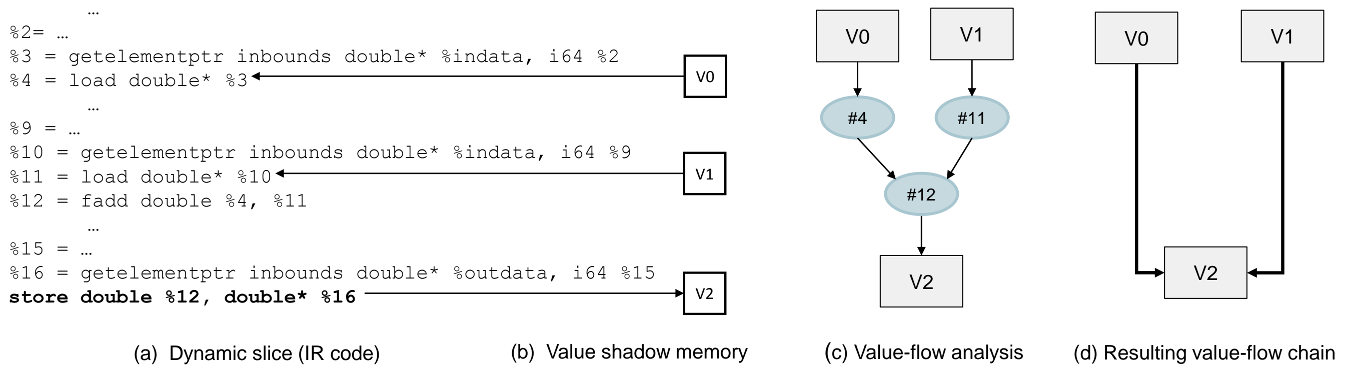



**Figure 7: Value flow-chain detection example**

method [2]. This slice is the set of dynamic IR (intermediate representation) instructions involved in the computation of the value ($v$) being stored in the memory. To determine those values on which the value ($v$) depends, CommAnalyzer traces it back inspecting the instructions and the registers along the data dependence edge of dynamic slice [1]. such a dependence backtracking continues for each data dependence edge until it encounters another value that has been recognized using the shadow memory (*MemVal*); whenever a new value is found, CommAnalyzer keeps it in the *Values* set (line 1 of Algorithm 2). Here, such a value turns out to be used in the computation of the value ($v$) being stored; it is said the former flows to the latter while the former is called a *source* value. At Line 5 of Algorithm 2, CommAnalyzer calculates $s$ which is a set of all the *source* values over the slice of $v$ being stored.

If such a value ($v$) does not exist in the *Values* set (i.e., new value), CommAnalyzer adds the value-flow chains between $v$ and $s$ to the set of the program value-flow chains ($\mathbb{VAL_FC}$) and updates the *Values* set; otherwise, $v$ is not unique and already exists in *Values*, and the store instruction just replicates this value and writes it to a new memory location. Usually, the value replication happens when the dynamic slice does not contain any compute instruction (e.g., direct load-store relation). Finally, CommAnalyzer updates the shadow memory *MemVal* with the new value. In addition, at the end of the program execution, CommAnalyzer performs the same value-flow chain detection for the return variables.

Figure 7 shows a simple example of the value-flow chain detection. For brevity, the example shows the thin dynamic slice [44] instead of the actual dynamic slice, i.e., it excludes the instructions that manipulate the memory pointers. After CommAnalyzer generates the dynamic slice of the target store instruction (a), it uses the shadow memory (b) to track the program values in the memory locations, and then it detects the value-flow in the dynamic slice. Since the exact memory addresses are available in the dynamic slice, CommAnalyzer inspects the shadow memory and records that registers *#4* and *#11* have values *V0* and *V1*, respectively. Next, a new value is computed in register *#12* using the values *V0* and *V1*, and the target store instruction writes this value to the memory. Finally, CommAnalyzer adds the new value-flow chain *(V2, {V0, V1})* in the set of the value-flow chains $\mathbb{VAL_FC}$.

[1] The outgoing edges of circle nodes in Figure 7 correspond to data dependence edges.

## 4.2 Communication Cost Estimation

*4.2.1 Value Communication Graph Formation.* After the detection of the value-flow chains between the input values, intermediate values, and output values, CommAnalyzer creates the *value communication graph* $VCG(V, E)$, where $V$ is a set of vertices that represent the program values and $E$ is a set of edges that represents the value-flow chains. This is achieved by simply generating a communication edge between the values in each value-flow chain. For example, at Line 5 of Algorithm 2, a connection edge is made between $v$ and every source value in $s$. $VCG$ is a directed and weighted graph, where the weight $w$ of the communication edge represents the number of times the sink (destination) value uses the source value.

**Value Graph Compression**. CommAnalyzer makes use of the value liveness [32] information $\mathbb{VAL_LIVE}$, which is generated during the dynamic value analysis, to reduce the communication graph size. An intermediate value is killed when it does not exist in the registers or the program memory. CommAnalyzer coalesces the vertices of the intermediate values that share the same memory location at non-overlapping liveness intervals into a single vertex. Such values are multiple exclusive updates to a single memory location. Finally, after the graph compression, CommAnalyzer updates the vertex and edge sets of the value communication graph.

**Singleton Detection and Removal**. The singleton value vertices appear in most scientific applications and are characterized by extreme connectivity, i.e., massive number of incoming and/or outgoing communication edges. Example of these values are the simulation coefficients, parameters, time steps, and error data. To reduce the communication on distributed-memory architectures, the singleton values should not be mapped to a specific compute node. Instead, each compute node/process creates a local copy of the singleton items and uses inter-process communication to exchange their values once per the simulation execution and/or the simulation time step. In fact, automated singleton detection and removal is well-known in large-scale circuit simulations [19, 46]; it reduces the overall communication by automatically detecting and duplicating the power/clock circuit nodes in each process. Similarly, CommAnalyzer adopts a threshold-based scoring approach for the singleton detection, which is a simplified variant of the proximity-based outlier detection methods [23].

**Algorithm 3** Singleton Detection Algorithm

**Input:** $\mathbb{VCG}$
**Output:** $\mathbb{S_VAL}$
1: $\mathbb{S_VAL}$ = {}
2: *HIGH* = 90%
3: **for each:** vertex $v \in \mathbb{VCG}$ **do**
4: *score* ← MAX( DensityS($\mathbb{VCG}$, $v$), DistanceS($\mathbb{VCG}$, $v$) )
5: **if** *score* ≥ *HIGH* **then**
6: $\mathbb{S_VAL} = \mathbb{S_VAL} \cup v$
7: **end if**
8: **end for**

Algorithm 3 shows the high-level singleton detection algorithm where the following definitions are used:

*Definition 4.2 (Value Degree Centroid).* The centroid is the minimum of the mean and the median value degree over the VCG, where the value degree of $v$ is defined as the number of value vertices adjacent to $v$ in the VCG.

*Definition 4.3 (Value Degree Distance).* The degree distance of a value *d(v)* is the distance between its degree and the centroid of the value degree cluster.

For each value vertex in VCG, CommAnalyzer computes the singleton score, which is the maximum of the density-based score and the distance-based score. When the singleton score is high, the value vertex is identified as a singleton. The density score of a value $v$ is estimated as the density of its row and column in the value adjacency matrix, while the distance score is computed by analyzing the value degree distribution of the VCG. If the degree of $v$ is larger than the degree centroid, the distance score of $v$ is computed as 1−(centroid/*d(v)*); otherwise, the distance score is zero. That is, the distance score estimates how positively far is $v$ from the centroid.

Once CommAnalyzer identifies the singleton vertices, it removes them from the communication graph, and maps the remaining values to the compute nodes using graph partitioning. Finally, it creates a local copy of the singletons in each compute node, and accounts for the singleton communication to project the total communication cost.

*4.2.2 Value Decomposition.* To predict the minimum communication volume across the compute nodes, CommAnalyzer maps the values to the compute nodes using a customized graph partitioning algorithm. The value mapping problem has three different optimization objectives: 1) maximizing the load balance which is estimated as the weighted sum of the vertices in each compute node, 2) minimizing the communication across the compute nodes which is estimated as the weighted sum of the edge cuts, and 3) generating connected value components in each compute node. CommAnalyzer uses the multilevel partitioning heuristics [29] to solve the value mapping problem in polynomial time, and then generates $\mathbb{VAL_PMAP}$ which maps each vertex in the value communication graph to a specific compute node. Figure 8 shows a simple example of the value decomposition using the graph clustering and partitioning algorithms. Finally, CommAnalyzer maps local copies of the singleton values to each compute node.

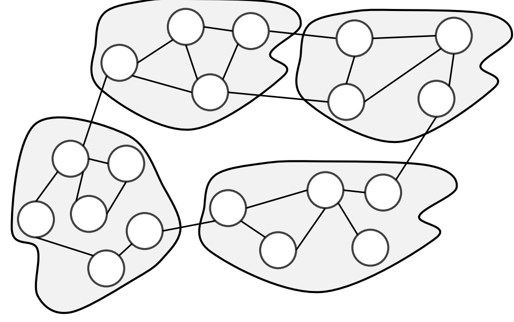


**Figure 8: Value decomposition using graph clustering and partitioning algorithms.**

*4.2.3 Communication Cost Estimation.* Once the value communication graph is decomposed, CommAnalyzer is ready to estimate the overall communication cost across the different nodes using the value-flow chains and the obtained value decomposition information.

**Algorithm 4** Communication Estimation Algorithm

**Input:** $\mathbb{VAL_FC}$, $\mathbb{VAL_PMAP}$, $\mathbb{S_VAL}$
**Output:** $\mathbb{COMM_COST}$
1: *commEdges* ← CommDetection($\mathbb{VAL_FC}$, $\mathbb{VAL_PMAP}$)
2: *commEdges* ← RedundantCommPruning(*commEdges*)
3: $\mathbb{COMM_MAT}$ ← CommMatGeneration(*commEdges*)
4: $\mathbb{COMM_MAT}$ ← SingletonComm($\mathbb{COMM_MAT}$, $\mathbb{S_VAL}$)
5: $\mathbb{COMM_COST}$ ← TotalCost($\mathbb{COMM_MAT}$)

Algorithm 4 shows the high level communication estimation algorithm. Once the program values are mapped to the different compute nodes, all the value-flow pairs with non-local values are identified as communication edges. Then, CommAnalyzer removes the redundant communication, by pruning the communication edges that carry the same value between the compute nodes, as the destination (sink) node needs to read this value once and later reuse the stored non-local values (See Figure 5). The final set of communication edges (after pruning) are used to update the corresponding entries in the communication matrix $\mathbb{COMM_MAT}$. An entry $(i, j)$ in the communication matrix represents the amount of the data transferred from the compute node $i$ to $j$ during the program execution.

Next, CommAnalyzer accounts for the singleton communication patterns to generate the final communication matrix. In particular, CommAnalyzer considers singleton values with only outgoing communication edges as collective read communication (broadcast), while singleton values with incoming and outgoing communication edges are collective read/write communication (Allreduce). Finally, CommAnalyzer estimates the communication cost as the overall cost of the communication matrix.

## 4.3 Implementation

We implemented the proposed approach for communication cost estimation on HPC clusters (explained above) using the LLVM

**Table 1: Target HPC benchmarks and applications**

| Application | Description | Input data |
|---|---|---|
| MatMul | Traditional matrix matrix multiplication | Three matrices of size $2048X2048$ |
| SPMV | Sparse matrix vector multiplication with Compressed Sparse Row (CSR) format | Matrix of size $4096X4096$ and 16M nonzero elements |
| MiniGhost [7] | Representative application of hydrodynamics simulation of solid materials in a Cartesian 3-D grid | 3D Grid of 16,777K points |
| Heat2D [36] | Canonical heat diffusion simulation in a homogeneous 2D space | 2D Grid of 16,777K points |
| LULESH [28] | The DARPA UHPC hydrodynamics challenge problem for Lagrangian shock simulation on an unstructured mesh | Mesh of 373 K elements |

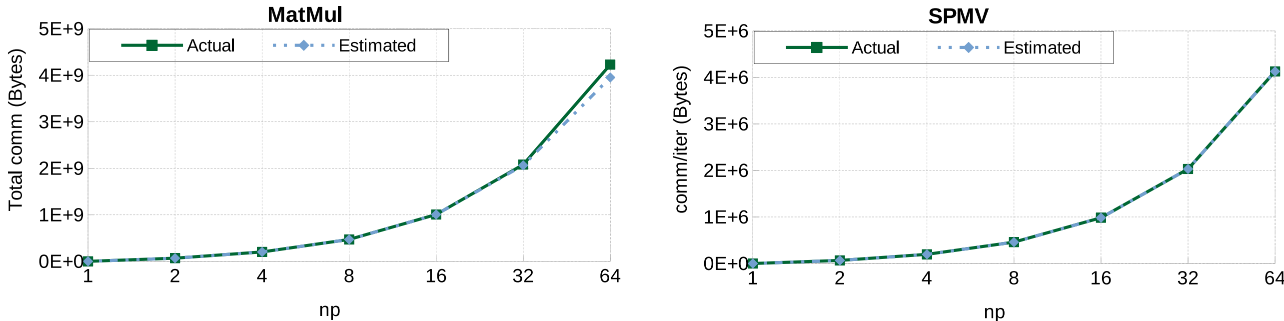



**Figure 9: Measured vs. estimated communication cost of MatMul and SPMV benchmarks**

compiler infrastructure [31]. CommAnalyzer uses the LLVM front-ends to parse the sequential C and FORTRAN code and to transform it to the LLVM Intermediate Representation (IR). The main dynamic analysis algorithm is implemented using the dynamic compiler LLI and works on the static single assignment (SSA) form of the LLVM IR. CommAnalyzer instruments the sequential IR and runs it using the provided input data set to estimate its communication cost on distributed-memory architectures.

## 5 CASE STUDIES

We illustrate the capabilities of CommAnalyzer using two benchmarks and three real-world HPC applications. Table 1 presents the applications considered in the case studies and the input data sets. The experiments use CommAnalyzer to analyze the sequential code of the target benchmarks and applications, and show its accuracy in comparison with the actual MPI+OpenMP implementations. The reported communication is for the core application kernels and ignores the initialization/setup and debugging code, and we use strong scaling where the total size of the working set is fixed on the different number of processes/nodes. The applications use double-precision floating point data types; however, the analysis works with any data type supported by LLVM.

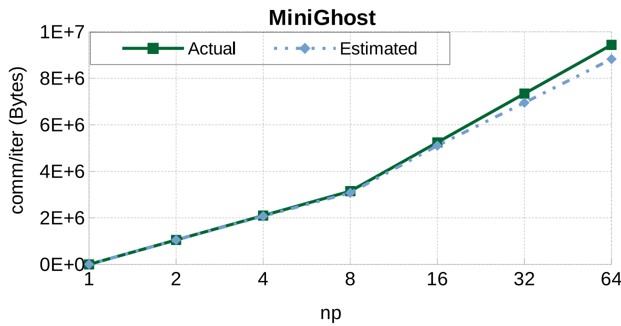
The test platform is a Linux cluster consists of 134 nodes, and each node contains two Intel Xeon E5-2680v3 processors running at 2.50 GHz. The nodes are connected with an EDR InfiniBand interconnection network. The platform uses batch queuing system that limits the number of nodes per user to 32 nodes. The test system uses CentOS Linux 7 distribution, and the applications are built using gcc 4.5 and OpenMPI 1.6.4. In the experiments, we launch one MPI process per processor and one OpenMP thread per core, and use the mpiP profiling tool [47] to measure the actual communication.

Our main goal is to show the feasibility of characterizing the inherent communication of sequential code with the value-flow chain (VFC) and identifying the values that need to be communicated when the code is ported to distributed-memory clusters (commEdges), regardless of the underlying data structures (regular or irregular). To show the accuracy of our proposed approach, we choose the communication cost (total bytes communicated) as the evaluation metric. While this metric is simple, it is important for estimating the asymptotic scalability/efficiency of a given program on distributed-memory architectures, which is bounded by the total work performed and the amount of data exchanged with different number of processes/nodes [15, 17].

### 5.1 Benchmarks

First, we evaluate the accuracy of CommAnalyzer using two canonical regular and irregular workloads: MatMul and SPMV. MatMul is a traditional matrix matrix multiplication. The MPI implementation of MatMul uses block-cyclic domain decomposition to distribute the matrices over the processes using the two-dimensional process topology. The traditional 2-D algorithm for matrix multiplication achieves linear communication scaling on distributed-memory architectures. However, the communication avoiding algorithms [26, 42] sustain higher performance, due to the sub-linear scaling of the communication time, with the cost of redundant computation/memory and higher programming complexity.

SPMV is an iterative sparse matrix vector multiplication that uses the compressed sparse row (CSR) format to store the input matrix. SPMV is an irregular application with indirect memory access



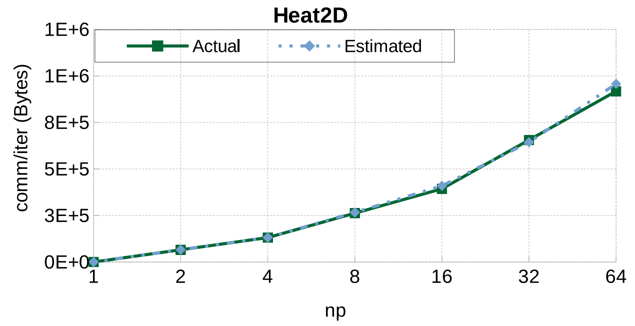



**Figure 10: Measured vs. estimated communication cost of MiniGhost and Heat2D**

pattern through multiple index arrays. To minimize the communication, the MPI implementation of SPMV uses one-dimensional domain decomposition to distribute row blocks of the sparse matrix and chunks of the RHS (input) and LHS (output) vectors. Each process computes a chunk of the output vector using its row block and the required elements of the RHS vector, which could exist in other processes. Therefore, the communication cost depends on the density of the input matrix. When the matrix is dense or semi-dense, each process requires $(np-1)$ remote chunks of the RHS, where $np$ is the number of processes, to compute its part of the LHS vector. Finally, the processes exchange the values of the LHS vector using collective communication.

Figure 9 shows the estimated communication cost in comparison with the actual communication for MatMul and SPMV. For the two benchmarks, CommAnalyzer achieves 98% prediction accuracy on average, and the maximum error is 7%. The communication cost of MatMul and SPMV scales linearly with the number of processes. In SPMV, CommAnalyzer detected $n$ singleton read/write items, where $n$ is the vector size (matrix dimension), corresponding to the output vector. In addition, due to the density of the input matrix, CommAnalyzer identified $n$ singleton read items which are the input vector. CommAnalyzer accounts for the singleton communication as collective communication (e.g., broadcast, Allreduce/-gather). Note, while the collective communication time usually scales sub-linearly with the number of processes (Log $np$), the total data transfer volume increases linearly.

## 5.2 MiniGhost and Heat2D

MiniGhost [6, 7] and Heat2D [36] are two regular HPC applications that use the structured grids design pattern, where a physical space is mapped to a Cartesian grid of points. Typically, the value of the grid points represents a material state such as the temperature, energy, and momentum.

MiniGhost is a representative Computational Fluid Dynamics (CFD) application for three-dimensional hydrodynamics simulation that models the flow and dynamic deformation of solid materials. Heat2D is a canonical heat transfer simulation that solves the Poisson partial differential equations of heat diffusion in a homogenous two-dimensional space. The two applications adopt the iterative finite-difference method with explicit time-stepping scheme to solve the simulation equations. Figure 11 shows the 3D 7-point and 2D 5-point finite-difference stencils used in MiniGhost and Heat2D, respectively.

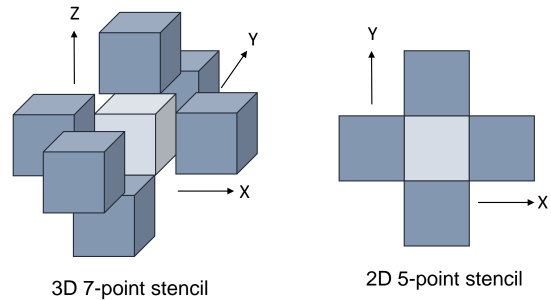



**Figure 11: The finite difference stencils used in MiniGhost and Heat2D**

Typically, the distributed-memory implementation of structured grids applications uses multi-dimensional domain decomposition and process topology to partition the global grid into multiple sub-grids, and to map each sub-grid to a specific process. To compute the values of the sub-grids in each time step, the processes exchange boundary elements (2-D faces and/or 1-D lines) with neighbors, which is known as halo exchange. Therefore, the communication volume depends on the surface area of the sub-grid boundaries and the total number of sub-grids (processes). At the end of the simulation, the value of the global grid is computed using collective communication (global reduction). In particular, MiniGhost and Heat2D use three-dimensional and two-dimensional domain decomposition/process topologies, respectively.

Figures 10 shows CommAnalyzer's estimation of the communication cost per iteration in comparison with the actual communication for MiniGhost and Heat2D. The results show that the prediction accuracy of CommAnalyzer is 97% and 98% on average for MiniGhost and Heat2D respectively, while the maximum error in both applications is 7%. The communication cost increases sub-linearly in the two applications which indicates the ability to scale to a large number of nodes. Moreover, for the same grid size, MiniGhost has an order-of-magnitude higher communication cost than Heat2D, due to the larger surface area of the sub-grids boundaries.

## 5.3 LULESH

LULESH [28] is a Lagrangian shock hydrodynamics simulation, and one of the five DARPA UHPC challenge problems. According to the the DoD High Performance Computing Modernization Program (HPCMP) Office [33], hydrodynamics simulation represents more than 30% of the DoD and DoE workloads. LULESH solves the Sedov blast wave problem [39] in three-dimensional space using an unstructured, hexa-hedral mesh. Each point in the mesh is a hexahedron with a center element that represents the thermodynamic variables (e.g., pressure and energy) and corner nodes that track the kinematic variables (e.g., velocity and position); Figure 12 shows the 3D view of a point in the mesh.

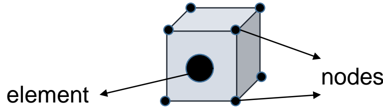



**Figure 12: LULESH uses an unstructured, hexa-hedral mesh with two centering.**

In the Lagrangian hydrodynamics simulation, the mesh follows the motion of the elements in the space and time. In particular, LULESH uses an explicit time stepping scheme (Lagrange leapfrog algorithm); in each time step, it advances the nodal variables, then updates the element variables using the new values of the nodal variables. To maintain the numerical stability, the next time step is computed using the physical constrains on the time step increment of all the mesh elements.

LULESH is a relatively large code with more than 40 computational kernels. It uses indirect memory access pattern via node and element lists and multiple-levels of workspace data structures. The MPI implementation of LULESH uses both cube process topology and domain decomposition (e.g., 3X3X3 topology) to distribute the mesh over the available processes, where each process can communicate with up to twenty-six neighbors. In each time step, there are three main communication operations. First, the processes exchange the node centered boundaries for the positions, acceleration, and force values of the mesh nodes. Second, they communicate the element centered boundaries for the velocity gradients. Third, a global collective communication is used to compute the next time step based on the physical constraints of all the mesh elements. In particular, the MPI implementation of LULESH has three different communication patterns: 3-D nearest neighbor, 3-D sweep, and collective broadcast and reduction [38].

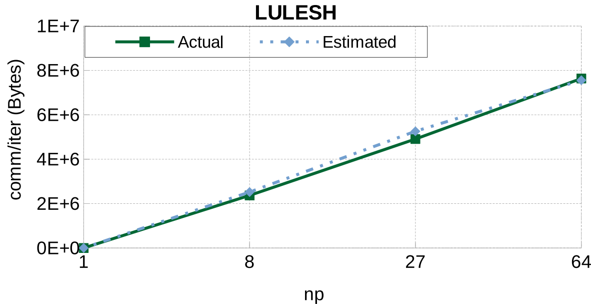



**Figure 13: Measured vs. estimated communication cost of LULESH**

**Table 2: Summary of the results**

| Application | Avg. Accuracy | Max Error % |
|---|---|---|
| Benchmarks | 98% | 7% |
| MiniGhost | 97% | 7% |
| Heat2D | 98% | 4% |
| LULESH | 95% | 7% |

Figures 13 describes the estimated and the actual communication costs for LULESH. In comparison with the actual communication, the prediction accuracy of CommAnalyzer is 95% on average while the maximum error is 7%. LULESH shows a good strong scaling trend; in particular, the communication cost increases by approximately a factor of 2 and 1.5, when the number of processes increases from 8 to 27 and from 27 to 64, respectively.

## 5.4 Summary

Table 2 summarizes the experimental results. The case studies demonstrate that CommAnalyzer achieves more than 95% prediction accuracy on average while the maximum error is less than 8%. The experiments represent the efficacy of CommAnalyzer for regular, irregular, and unstructured problems that have different communication patterns such as 2-D/3-D nearest neighbor, 3-D sweep/wavefront, 2-D broadcast, and collective broadcast/reduction communication.

# 6 RELATED WORK

This section first discusses the prior work on communication cost estimation from sequential code, and then explains how the proposed approach (CommAnalyzer) integrates with the orthogonal performance modeling and prediction approaches for distributed-memory architectures to advance the state of the art in this field.

## 6.1 Communication Cost Estimation

Several approaches have been proposed for communication cost estimation from sequential code to enable the automatic code generation for distributed-memory platforms [18, 22, 48]. The FORTRAN-D compiler [22] uses static data dependence analysis to detect the communication between the different sections of the data array, which is partitioned according to a user-specified decomposition technique. However, this approach suffers from communication overestimation, and it is limited to regular applications.

The SUIF compiler [48] solves the communication overestimation problem using exact static data-flow analysis, but it is only applicable to affine loop nests with regular memory access pattern. In contrast, CommAnalyzer predicts the communication between the program values regardless of the underlying data structures and without any user-specified decomposition techniques using a combination of novel dynamic analysis techniques and graph algorithms. Thus, CommAnalyzer is applicable to a wide range of regular and irregular applications.

While the above approaches estimate the communication cost using the SPMD execution mode (owner-computes), Bondhugula [8] presents a polyhedral framework to estimate the communication when there is no fixed data ownership, i.e. the data moves between

compute nodes according to the distribution of the computations and the data dependences. However, this approach is limited to regular applications with affine loop nests as well. Although the SPMD execution model is by far the dominant approach on HPC clusters, it would be interesting to extend our novel techniques to other execution models, where the data ownership is passed among the compute nodes.

## 6.2 Performance Prediction on HPC Clusters

There is a large body of prior work for performance modeling and prediction on distributed-memory architectures. Analytical performance modeling approaches provide useful insights into the application performance by modeling the interactions between the HPC platform and the application. However, they require tedious manual analysis of the target applications, hardware architectures, and interconnection networks to estimate the communication time and the overall performance. The LogP model family, e.g., LogP [12] and LogGP [3], contain several *high-level* analytical models that predict the communication time and the performance of MPI applications using a few parameters that abstract the application characteristics and hardware details.

ASPEN [43] is a modeling language for describing and developing formal application/machine models to explore the algorithm and architecture design options. PALM [45] simplifies the performance modeling process by providing a modeling annotation language to support the generation of standard analytical models from the annotated MPI source code. Snavely et al. [41] provide a performance prediction framework that automatically generates and evaluates analytical models for the application performance. These models are generated using the communication and memory traces of the MPI application and the abstract machine profiles.

Distributed-memory simulators, such as LogGOPSim [24] and DIMEMAS [37], incorporate detailed network and architecture models to estimate the communication time and the application performance. However, they require the MPI parallel implementation to profile the communication operations and generate the application trace. MUSA [16] adopts a multi-level simulation approach with different levels of hardware details, simulation cost, and simulation accuracy. In addition, it identifies and simulates representative application phases to reduce the overall simulation time.

The scalability analysis tools [5, 9, 11, 50] project the performance and the scaling behavior of a given MPI implementation at massive scale based on experiments on a small-scale platform. Usually, these tools extract the communication, computation, and/or memory traces of the MPI application using dynamic instrumentation and profiling of small-scale executions. Therefore, they require the distributed parallel implementations and at least a single node of the target cluster to predict the performance on multiple nodes. TAU [40] and HPC toolkit [1] are integrated frameworks for portable performance analysis, profiling, and visualization. Similar to scalability analysis tools, their main goal is to simplify the performance analysis and diagnosis of the existing MPI code, rather than estimating the potential performance at large scale before investing effort and time in developing the MPI implementation of the original applications.

Identifying the communication patterns of HPC applications is helpful in the design space exploration of both the system architectures and parallel algorithms. Several tools [30, 38] have been proposed for the detection of the MPI communication pattern by finding a match between the actual inter-process communication and a set of known communication templates/patterns. Usually, these tools instrument the MPI implementation to capture the inter-process communication and generate the communication matrix across the MPI processes, and then they recognize the existing communication pattern in the communication matrix. In particular, AChax [38] can identify multiple communication patterns (such as nearest neighbor, sweep/wavefront, broadcast, and reduction) in the communication matrix and generate a parametrized communication model for the actual communication based on a combination of the detected communication patterns.

CommAnalyzer advances the current state of the art by automatically estimating the communication cost from the sequential code rather than the MPI implementation. As such, it enables the above performance prediction, scalability analysis, and communication pattern detection tools to support a wide range of application, algorithm, and system developers/designers. In particular, we expect that CommAnalyzer integrates well with existing scalability analysis tools to project the performance at large scale using small-scale analysis and profiling runs of the sequential code rather than the MPI code.

# 7 CONCLUSION

In this paper, we proposed a novel tool to detect the inherent communication pattern of the sequential code regardless of the underlying data structures, and to predict the communication cost of such code when executed on multiple compute nodes using the SPMD execution model. We implemented CommAnalyzer in the LLVM compiler framework and used novel dynamic analysis and graph partitioning algorithms to estimate the minimum communication volume on distributed-memory architectures. The experiments demonstrate the efficacy of CommAnalyzer for regular, irregular, and unstructured problems with different communication patterns. The results show that CommAnalyzer is highly accurate in estimating the communication cost on HPC clusters across two benchmarks and three real-world applications. It achieved more than 95% prediction accuracy on average while the maximum error is less than 8%.

CommAnalyzer not only helps the users to project the communication cost of MPI applications before even developing one, but also enables the application and system designers to explore the algorithm-architecture design space at the early stages of the development process. There remain many opportunities to expand on the proposed tool. Integrating CommAnalyzer with existing scalability analysis tools allows the users to project the large-scale performance of their applications using small-scale experiments with the sequential code rather than the MPI code. In addition, it enables the estimation of the optimal number of the required compute nodes for each application to achieve high HPC system utilization. Moreover, CommAnalyzer is a perfect candidate to drive a communication-aware workload distribution scheme to efficiently utilize the available compute resources across HPC nodes [20].

## REFERENCES


[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.

[3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. Loggp: Incorporating long messages into the logp model—one step closer towards a realistic model for parallel computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.

[4] A. Bahmani and F. Mueller. Scalable performance analysis of exascale mpi programs through signature-based clustering algorithms. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 155–164. ACM, 2014.

[5] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. De Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 368–377. ACM, 2008.

[6] R. F. Barrett, S. D. Hammond, C. T. Vaughan, D. W. Doerfler, M. A. Heroux, J. P. Luitjens, and D. Roweth. Navigating an evolutionary fast path to exascale. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 355–365, 2012.

[7] R. F. Barrett, C. T. Vaughan, and M. A. Heroux. Minighost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing. Technical report, 2011.

[8] U. Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 33. ACM, 2013.

[9] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 45. ACM, 2013.

[10] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to upc and language specification. Technical report, 1999.

[11] M. Casas, R. Badia, and J. Labarta. Automatic analysis of speedup of mpi applications. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 349–358. ACM, 2008.

[12] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[13] R. E. Diaconescu and H. P. Zima. An approach to data distributions in chapel. *International Journal of High Performance Computing Applications*, 21(3):313–335, 2007.

[14] J. Diaz, C. Munoz-Caro, and A. Nino. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on parallel and distributed systems*, 23(8):1369–1386, 2012.

[15] A. Grama, A. Gupta, and V. Kumar. Isoefficiency function: A scalability metric for parallel algorithms and architectures. In *IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice*. Citeseer, 1993.

[16] T. Grass, C. Allande, A. Armejach, A. Rico, E. Ayguadé, J. Labarta, M. Valero, M. Casas, and M. Moreto. Musa: a multi-level simulation approach for next-generation hpc machines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 45. IEEE Press, 2016.

[17] A. Gupta and V. Kumar. Performance properties of large scale parallel systems. *Journal of Parallel and Distributed Computing*, 19(3):234–244, 1993.

[18] M. Gupta and P. Banerjee. Compile-time estimation of communication costs on multicomputers. In *Parallel Processing Symposium, 1992. Proceedings., Sixth International*, pages 470–475. IEEE, 1992.

[19] A. E. Helal, A. M. Bayoumi, and Y. Y. Hanafy. Parallel circuit simulation using the direct method on a heterogeneous cloud. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015.

[20] A. E. Helal, P. Sathre, and W.-c. Feng. Metamorph: A library framework for interoperable kernels on multi- and many-core clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 11:1–11:11, November 2016.

[21] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 3, 2009.

[22] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling fortran d for mimd distributed-memory machines. *Communications of the ACM*, 35(8):66–80, 1992.

[23] V. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial intelligence review*, 22(2):85–126, 2004.

[24] T. Hoefler, T. Schneider, and A. Lumsdaine. Loggopsim: Simulating large-scale applications in the loggops model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 597–604, New York, NY, USA, 2010. ACM.

[25] R. Hornung, J. Keasler, and M. Gokhale. Hydrodynamics challenge problem. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2011.

[26] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[27] N. Jain, A. Bhatele, M. P. Robson, T. Gamblin, and L. V. Kale. Predicting application performance using supervised learning on communication features. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 95. ACM, 2013.

[28] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, et al. Exploring traditional and emerging parallel programming models using a proxy application. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 919–932. IEEE, 2013.

[29] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–13, Washington, DC, USA, 1998. IEEE Computer Society.

[30] D. J. Kerbyson and K. J. Barker. Automatic identification of application communication patterns via templates. *ISCA PDCS*, 5:114–121, 2005.

[31] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. pages 75–86, 2004.

[32] S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, 1997.

[33] D. H. P. C. M. P. Office. Department of defense high performance computing modernization program fy 2010 requirements analysis report. Technical report, DTIC Document, 2010.

[34] OpenACC Working Group and others. The OpenACC Application Programming Interface, 2011.

[35] OpenMP, ARB. Openmp 4.0 specification, May 2013.

[36] N. Ozisik. *Finite difference methods in heat transfer*. CRC press, 1994.

[37] G. Rodriguez, R. Badia, and J. Labarta. Generation of simple analytical models for message passing applications. In *Euro-Par 2004 Parallel Processing*, pages 183–188. Springer, 2004.

[38] P. C. Roth, J. S. Meredith, and J. S. Vetter. Automated characterization of parallel application communication patterns. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 73–84. ACM, 2015.

[39] L. Sedov. Similarity and dimensional methods in mechanics, acad. *Press, New York*, 1959.

[40] S. S. Shende and A. D. Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[41] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 21–21. IEEE, 2002.

[42] E. Solomonik and J. Demmel. Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. In *European Conference on Parallel Processing*, pages 90–109. Springer, 2011.

[43] K. L. Spafford and J. S. Vetter. Aspen: A domain specific language for performance modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 84:1–84:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[44] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 112–122, New York, NY, USA, 2007. ACM.

[45] N. R. Tallent and A. Hoisie. Palm: Easing the burden of analytical performance modeling. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 221–230, New York, NY, USA, 2014. ACM.

[46] H. K. Thornquist, E. R. Keiter, R. J. Hoekstra, D. M. Day, and E. G. Boman. A parallel preconditioning strategy for efficient transistor-level circuit simulation. In *Computer-Aided Design-Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pages 410–417. IEEE, 2009.

[47] J. Vetter and C. Chambreau. mpip: Lightweight, scalable mpi profiling. *URL http://mpip.sourceforge.net*.

[48] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, et al. Suif: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices*, 29(12):31–37, 1994.

[49] X. Wu, F. Mueller, and S. Pakin. Automatic generation of executable communication specifications from parallel applications. In *Proceedings of the international conference on Supercomputing*, pages 12–21. ACM, 2011.

[50] J. Zhai, W. Chen, and W. Zheng. Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. In *ACM Sigplan Notices*, volume 45, pages 305–314. ACM, 2010.