CommAnalyzer: Automated Estimation of Communication Cost and Scalability on HPC Clusters from Sequential Code

Ahmed E. Helal[1], Changhee Jung[2], Wu-chun Feng[1,2], Yasser Y. Hanafy[1]
Electrical & Computer Eng.[1] and Computer Science[2], Virginia Tech
{ammhelal,chjung,wfeng,yhanafy}@vt.edu

ABSTRACT

To deliver scalable performance to large-scale scientific and data analytic applications, HPC cluster architectures adopt the distributed-memory model. The performance and scalability of parallel applications on such systems are limited by the communication cost across compute nodes. Therefore, projecting the minimum communication cost and maximum scalability of the user applications plays a critical role in assessing the benefits of porting these applications to HPC clusters as well as developing efficient distributed-memory implementations. Unfortunately, this task is extremely challenging for end users, as it requires comprehensive knowledge of the target application and hardware architecture and demands significant effort and time for manual system analysis.

To streamline the process of porting user applications to HPC clusters, this paper presents CommAnalyzer, an automated framework for estimating the communication cost on distributed-memory models from sequential code. CommAnalyzer uses novel dynamic program analyses and graph algorithms to capture the inherent flow of program values (information) in sequential code to estimate the communication when this code is ported to HPC clusters. Therefore, CommAnalyzer makes it possible to project the efficiency/scalability upper-bound (i.e., Roofline) of the effective distributed-memory implementation before even developing one. The experiments with real-world, regular and irregular HPC applications demonstrate the utility of CommAnalyzer in estimating the minimum communication of sequential applications on HPC clusters. In addition, the optimized MPI+X implementations achieve more than 92% of the efficiency upper-bound across the different workloads.

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, 18]. 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 [43], to exploit the available compute resources seamlessly both within a node and across nodes. Alternatively, the partitioned global address space (PGAS) programming models (e.g., Chapel [17] and UPC [13]) abstract explicit communication using distributed memory objects; however, the user still needs to manage the data distribution and locality across compute nodes.

Current (and future) HPC cluster architectures suffer from an increasing gap between the computation and communication costs, i.e., the cost of data transfers can be orders of magnitude higher than the cost of compute operations [8]. Therefore, the application scaling on HPC clusters is limited by the communication cost across 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 [21, 23].

Hence, fast and accurate prediction of the communication cost of user applications would provide many benefits, including projecting the potential speedup on HPC clusters, constructing bound-and-bottleneck scaling models, and guiding the development and optimization of distributed-memory implementations. Unfortunately, estimating the communication and scalability of a given application is a complex and time-consuming process that requires extensive manual analysis and a wide array of expertise in the application domain, HPC architecture, and programming model.

Researchers have created several scalability analyzers [5, 11, 14, 22, 58, 59] and communication pattern detectors [4, 36, 47] to study the effect of the data transfers on the application performance and to provide valuable insights on the optimization of the communication bottlenecks. However, these tools are limited only to *MPI implementations*. That is, the estimated communication is specific to the *given MPI implementation* and its workload decomposition/distribution strategy rather than the inherent characteristics of the *original application*. Moreover, due to the parallel programming effort and time on distributed-memory systems, the MPI implementation is often not available in the early stages of the development process. Thus, there is a compelling need for automated tools, that can analyze the sequential applications and predict the communication cost of their parallel execution on HPC clusters, to estimate the scalability and performance beforehand without needing comprehensive knowledge of the applications and cluster architectures.

1.1 CommAnalyzer Framework

This paper presents CommAnalyzer, an automated framework for communication cost estimation from sequential code, to figure out the scaling characteristics of running the code in parallel on a distributed system using the single program, multiple data (SPMD) execution model. Figure 1 shows the proposed framework that

HPDC '18, June 11–15, 2018, Tempe, AZ, USA

ACM ISBN 978-1-4503-5785-2/18/06…$15.00
https://doi.org/10.1145/3208040.3208042

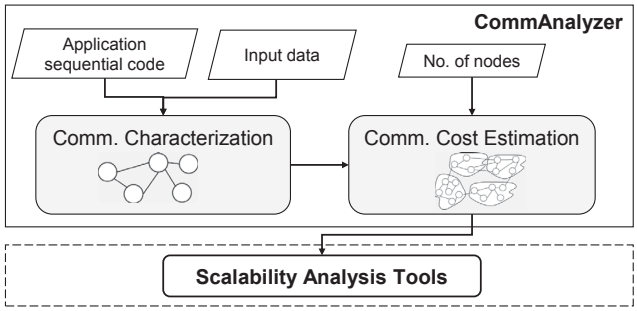


Figure 1: Overview of CommAnalyzer framework

takes as inputs the sequential application code (written in any of the languages supported by LLVM compiler [40]), representative input data, and the number of compute nodes (e.g., 2-64 nodes).

The key idea is to reformulate the problem of estimating the communication of a parallel program into that of analyzing the inherent flow of program values (information) in the sequential application. Hence, CommAnalyzer uses novel dynamic program analyses that build a *value communication graph* (VCG) from sequential code taking into account the data-flow behavior of its program values. Since parallel programs are typically optimized to minimize the communication between compute nodes, their data transfers are likely to serve as a cut for partitioning the VCG. CommAnalyzer in turn leverages graph partitioning algorithms over the VCG to automatically identify sender/receiver entities and to estimate their communication. As a result, for a given sequential application, CommAnalyzer allows the user to project the performance upper-bound of its effective SPMD execution, regardless of the target programming model (e.g., MPI and PGAS).

The communication cost estimation of CommAnalyzer can be used by scalability analysis tools, such as Extra-P [11], to estimate the strong and weak scaling of the communication. These tools perform regression analysis using the scaling functions (regression hypothesis) that exist in HPC applications to estimate a user-specified metric, e.g., FLOPs, communication, etc., at a large scale from a set of small-scale measurements or predictions with different problem sizes and/or number of nodes. Furthermore, CommAnalyzer makes it possible to construct bound-and-bottleneck models of the parallel efficiency and scalability on distributed-memory clusters.

Use Cases. CommAnalyzer accelerates the application and system design in the early stages of the development process by allowing end users to figure out the scaling behavior of their sequential applications. As such, domain-scientists can quickly make informed decisions about the need to explore other solutions/algorithms to the problem at hand to attain better parallel performance. That way, CommAnalyzer enables the system designers to evaluate the HPC system design alternatives to achieve the required performance.

Even if a distributed-memory implementation of the target application is available, CommAnalyzer still plays a critical role in the optimization process by generating bound-and-bottleneck scaling models that show how close the current parallel implementation is to the scalability/efficiency Roofline. In addition, by estimating the communication from the sequential code rather than the MPI parallel code, CommAnalyzer empowers existing scalability analysis tools for MPI applications to serve a wide range of end users.

1.2 Contributions

Unlike previous approaches that are limited by the imprecision of *compile-time* analyses [24, 30, 56], CommAnalyzer proposes a novel *dynamic* analysis approach that instruments the sequential code to precisely capture the runtime information required to detect not only static value-flow (communication) dependencies, but also dynamic value-flow dependencies through multiple levels of access indirection. Thus, CommAnalyzer is applicable for both regular and irregular problems and works also for programs that cannot be auto-parallelized. The following are the contributions of this work:

- *A novel and automated approach for estimating the communication cost of sequential applications when ported to HPC clusters based on value-flow analysis, value liveness analysis, dynamic program slicing, and graph algorithms.* This approach is applicable for regular, irregular, and unstructured problems. Using the estimated communication, we can successfully project the efficiency upper-bound of the effective SPMD implementations on distributed-memory HPC clusters (Sections 4 and 5).
- *Model validation and case studies using both regular and irregular workloads: matrix multiplication and sparse matrix vector multiplication, as well as four structured and unstructured representative applications: MiniGhost [7], Heat2D [44], LULESH [37], and K-means [39].* The experiments demonstrate the utility of CommAnalyzer in identifying the minimum communication cost on HPC clusters with more than 95% accuracy on average and show that the optimized MPI+OpenMP implementations can attain more than 92% of the efficiency upper-bound (Section 6).

2 BACKGROUND

2.1 Distributed-Memory Execution Model

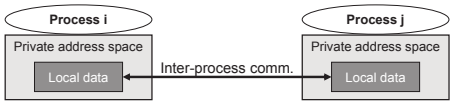


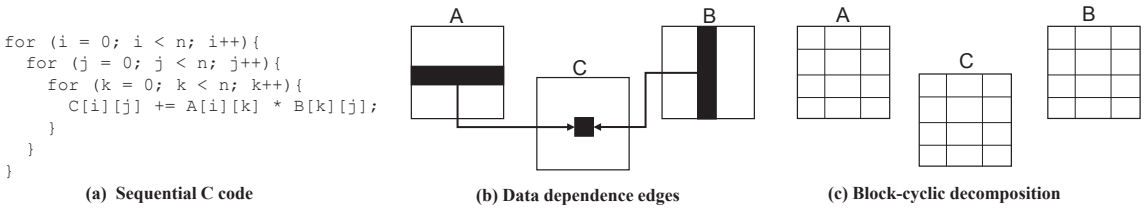
Figure 2: The SPMD execution model

We assume that the applications running on the target HPC clusters follow the single program, multiple data (SPMD) execution model, which is the dominant approach on such architectures [6, 18]. Figure 2 shows the SPMD execution model, where the program data is partitioned and mapped to different processes (compute nodes) and all processes execute the same program to perform computations on their data (i.e., owner-computes rule). The data owned by each process is stored on its private (local) address space, and when the local computations on a process involve non-local data, this data is accessed using inter-process communication.

2.2 Simple Estimation of Communication Cost

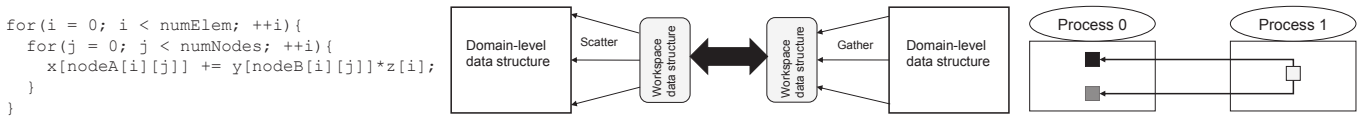
In SPMD execution, the communication results from dependencies between local and non-local data. Therefore, a simple approach for estimating the communication cost is to partition the program data and then identify the data dependencies across different partitions.

Figure 3 shows a simple communication estimation for matrix multiplication on distributed-memory architectures. The sequential C code (a) is given to the compiler, and a traditional data-flow analysis is used to identify the data dependence [41] (b) between the data items. Next, a domain decomposition method such as



(a) Sequential C code (b) Data dependence edges (c) Block-cyclic decomposition

Figure 3: Simple estimation of communication cost for matrix multiplication



(a) Indirect memory access (b) Workspace mask the true data dependencies (c) Redundant communication

Figure 4: Challenges for estimating the communication cost

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 nodes. This approach has been used by the auto-parallelizing compilers for distributed-memory architectures [30, 56] in the early days of HPC. While this simple communication analysis is sufficient for regular and structured problems such as matrix multiplication, real-world scientific applications with irregular computation and unstructured memory access patterns are a lot more challenging and therefore require sophisticated analysis techniques.

3 CHALLENGES

3.1 Indirect Memory Access

Irregular and unstructured problems arise in many important scientific applications such as hydrodynamics [29, 35]. These problems are characterized by an indirect memory access pattern via index data structures, which cannot be determined at compile time. Figure 4(a) shows an example of such an indirect access. Here, 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. Thus, it is impossible to figure out which part of x (y) is defined (used) due to the lack of runtime information.

3.2 Workspace Data Structures

In unstructured problems, it is common to gather data items from the program-level (domain-level) data structures into workspace (temporary) data structures. The actual computations are performed in the workspace, and then the 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(b) shows an example of an 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 a massive number of local data dependence edges between the program-level data and the workspace data, which results in inaccurate estimation of the actual communication cost.

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 data items that use the output of these computations (e.g., simulation error and time step). 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 reduce) to update their values at the beginning and 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.

3.4 Redundant Communication

Computing the communication cost as the number of data dependence edges between data items that exist in different processes is subject to overestimation. Figure 4(c) 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 (instead of two times) 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.

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 distributed-memory parallel codes. CommAnalyzer relies on the following observation; HPC developers always optimize (minimize) the communication of their distributed-memory parallel programs 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, which would otherwise break the program correctness. As a result, analyzing the data communication

in the sequential codes is able to serve as a basis for estimating the communication cost of their distributed-memory implementations. 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 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 heredity 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 the *value communication graph* (VCG).

Algorithm 1 CommAnalyzer algorithm

Input: **PROGRAM**, **N**
Output: COMM_COST
1: VAL_FC ← ValFcDetection(**PROGRAM**)
2: VAL_LIVE ← ValLiveAnalysis(**PROGRAM**)
3: VCG ← CommGraphFormation(VAL_FC, VAL_LIVE)
4: VAL_PMAP ← ValDecomposition(VCG, **N**)
5: COMM_COST ← CommEstimation(VAL_FC, 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 then 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 the memory during program execution.

CommAnalyzer also analyzes the live range [41] (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: *input, output, and 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 program 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 memory locations only 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 an *intermediate* value. In addition, any program value that exists only in registers is treated as an *intermediate* value.

Algorithm 2 Value-flow chain detection algorithm

Input: **PROGRAM**
Output: VAL_FC
1: *Values* = {} ▷ program values
2: *MemVal* = {} ▷ shadow memory-to-value map
3: **for each:** dynamic store (write) operation w **do**
4: DS ← DynamicSlicing(w, **PROGRAM**)
5: v, s ← ValueFlowAnalysis(DS, *MemVal*)
6: **if** $v \notin$ *Values* **then** ▷ writing new value
7: *Values* = *Values* ∪ v
8: VAL_FC = VAL_FC ∪ (v , s)
9: **end if**
10: *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). 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. Thus, each shadow memory location tracks the latest value stored in the corresponding location in the original memory. Since CommAnalyzer works on the static single assignment (SSA) form [41] of the sequential code (e.g., the LLVM IR [40]), 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 method [2]. This slice is the set of dynamic IR 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

[1]The outgoing edges of circle nodes in Figure 5 correspond to data dependence edges.

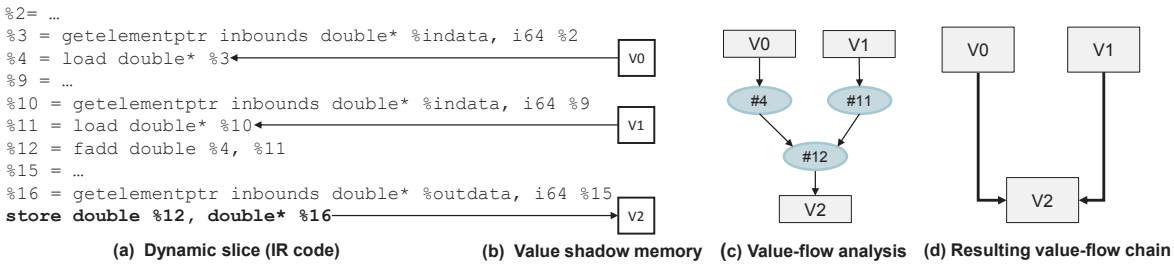


Figure 5: Value-flow chain detection example

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, the found 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 the 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 (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 instructions (e.g., direct load-store relation). Finally, CommAnalyzer updates the shadow memory *MemVal* with the new value. Further, at the end of the program execution, CommAnalyzer performs the same value-flow chain detection for the return variables.

Figure 5 shows a simple example of the value-flow chain detection. For brevity, the example shows the thin dynamic slice [52] 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 VAL_FC.

4.2 Communication Cost Estimation

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

Value Graph Compression. CommAnalyzer utilizes the value liveness [41] (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. This vertex coalescing does not impose artificial constraints on the program data partitioning according to the SPMD execution model, where processes compute all the intermediate values for the data items (memory locations) that they own. Finally, after the graph compression, CommAnalyzer updates the weights of the new vertex and edge sets of the value communication graph $VCG(V, E)$ to account for the communication cost before graph compression.

Singleton Detection and Removal. The singleton values 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 simulation coefficients, parameters, and time step. 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 global 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 [25, 54]; 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 [31].

Algorithm 3 Singleton Detection Algorithm

Input: VCG
Output: S_VAL
1: S_VAL = {}
2: **for each:** vertex $v \in$ VCG **do**
3: *score* ← MAX(DENSITYS(VCG, v), DISTANCES(VCG, v))
4: **if** *score* ≥ *HIGH* **then** ▷ *HIGH* = 90%
5: S_VAL = S_VAL ∪ v
6: **end if**
7: **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 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 v is 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 global communication to project the total communication cost.

4.2.2 Value Decomposition. To predict the minimum communication across compute nodes, CommAnalyzer maps the values to the 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 the weighted sum of the edge cuts, and 3) generating connected value components in each compute node. CommAnalyzer uses the multilevel partitioning heuristics [38] to solve the value mapping problem in polynomial time, and then generates VAL_PMAP which maps each vertex in the value communication graph to a specific compute node. Finally, CommAnalyzer maps local copies of the singleton values to each compute node.

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

Algorithm 4 Communication Estimation Algorithm

Input: VAL_FC, VAL_PMAP, S_VAL
Output: COMM_COST
1: *commEdges* ← CommDetection(VAL_FC, VAL_PMAP)
2: *commEdges* ← RedundantCommPruning(*commEdges*)
3: COMM_MAT ← CommMatGeneration(*commEdges*)
4: COMM_MAT ← SingletonComm(COMM_MAT, S_VAL)
5: COMM_COST ← TotalCost(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 4(c)). The final set of communication edges (after pruning) are used to update the corresponding entries in the communication matrix COMM_MAT. An entry (i, j) in the communication matrix represents the amount of the data transferred from a compute node i to j during the program execution. Next, CommAnalyzer accounts for the singleton communication patterns to generate the final communication matrix. While there are several options to synchronize the singleton values, such as broadcast, reduce, allgather, and allreduce, the minimum communication cost to synchronize a singleton value is $O(p-1)$, where p is the number of compute nodes. In particular, source (input), sink (output), and reduction (input/output) singleton values require at least $p-1$, $p-1$, and $2(p-1)$ communication words, respectively. CommAnalyzer uses this lower bound to account for the singleton communication cost.

Finally, CommAnalyzer estimates the communication cost as the overall cost of the communication matrix and computes the communication intensity of the program. The communication intensity of an application is $I_c = C/W$, where C is the communication cost in bytes and W is the work in FLOPs. CommAnalyzer computes W by counting the number of floating-point IR instructions during the program instrumentation.

4.3 Implementation and Complexity

We implemented the proposed approach for communication cost estimation on HPC clusters (explained above) using the LLVM compiler infrastructure [40]. CommAnalyzer uses the LLVM front-ends to parse the sequential 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 with the provided input data set on top of LLI, to estimate its communication cost.

The time complexity of the instrumentation (Section 4.1) is linear, as generating the dynamic program slice using the Reduced Dynamic Dependence (RDD) graph method requires a fixed number of operations for each dynamic instruction. The memory complexity of the instrumentation depends on the inherent communication of the sequential code. For embarrassingly-parallel code, the memory complexity is linear (shadow memory), and if the code is fully connected (each value depends on all other values), the memory complexity is quadratic in the worst case (shadow memory and adjacent values). However, in practice, each value depends on a limited number of neighboring values, i.e., the adjacency matrix representation of VCG (Value Communication Graph) is sparse even for structured and dense code (see Section 6). Note that the number of vertices of VCG is bounded by the number of memory words as the intermediate values in a single memory location are stored as one vertex in VCG.

The offline analysis (Section 4.2) has a time complexity of $O((|V|+|E|)\log p)$ and a memory complexity of $O(|V|+|E|)$, where V and E are the sets of vertices and edges of VCG, and p is the number of distributed-memory compute nodes.

Table 1: Target HPC benchmarks and applications

Application	Description	Input data
MatMul	Traditional matrix multiplication	Three matrices of size 4K×4K
SPMV	Sparse matrix vector multiplication with Compressed Sparse Row (CSR) format	Dense square matrix of order 32K
MiniGhost [7]	Representative application of hydrodynamics simulation of solid materials in a Cartesian 3D grid	3D Grid of size 1K×1K×1K
Heat2D [44]	Canonical heat diffusion simulation in a homogeneous 2D space	2D Grid of size 32K×32K
LULESH [37]	The DARPA UHPC hydrodynamics challenge problem for Lagrangian shock simulation on an unstructured hexa-hedral mesh	Strong Scaling Problem: Mesh of size 240^3, Weak Scaling Problem: Mesh of size 120^3
K-means [39]	Unsupervised machine-learning application for data clustering	10M data objects in 34D features space clustered into 5 groups

5 EFFICIENCY ROOFLINE MODEL

To show the importance of estimating the communication cost and intensity (I_c) of a given sequential application, this section proposes a high-level model to project the efficiency upper-bound on distributed-memory architectures. Our analysis is inspired by the Roofline model [55] which shows that the performance on shared-memory systems is bounded by the computation intensity of the application, the memory bandwidth, and the floating-point throughput, and ignores several overheads such as the memory access latency, parallel/scheduling overhead, and resource contention. Such simple bound and bottleneck analysis is useful for projecting the performance in the early stages of the development process and for optimizing the actual parallel implementation to minimize the gap between the upper-bound and the achieved performance. Section 6 shows the effectiveness of the proposed models for projecting the performance upper-bound of real-world HPC workloads.

In the analysis, we use the following notations:

- n, p: the problem size and the number of compute nodes.
- $T_1(n)$: the single-node execution time.
- $T_o(n, p)$: the parallel overhead function.
- $E(n, p)$: the parallel efficiency, i.e. $speedup/p$.

According to the classical isoefficiency analysis [21, 23], the asymptotic efficiency of a given application on distributed-memory architectures is determined by the growth of the total work performed and the amount of data exchanged as a function of the problem size and the number of nodes. In particular, the parallel efficiency function is given by [2]:

$$E(n,p) = 1/(1 + \frac{T_o(n,p)}{T_1(n)}) \quad (1)$$

To project the efficiency upper-bound $E_u(n, p)$, we estimate the lower-bound on the overhead function as follows:

$$T_{o,l}(n,p) = \frac{C \times (1 - O_c)}{B_c} \quad (2)$$

where C is the communication in bytes, B_c is the network bisection bandwidth, and O_C is the maximum communication overlap.

The single-node execution time can be formulated as:

$$T_1(n) = W/R \quad (3)$$

Where W is the work in FLOPs, and R is the single-node throughput.

By substituting equations 2 and 3 for $T_o(n, p)$ and $T_1(n)$ in equation 1, and using the communication intensity definition (i.e., $I_c = C/W$), the efficiency upper-bound is computed as:

$$E_u(n,p) = 1/(1 + \frac{R \times I_c \times (1 - O_c)}{B_c}) \quad (4)$$

[2] The readers can refer to [21] for understanding how equation 1 is derived from the ratio of the speedup to the number of nodes.

As such, this equation binds the efficiency upper-bound, the single-node throughput, the communication intensity, and the network bandwidth. The communication intensity I_c is obtained by CommAnalyzer, while the single-node throughput can be estimated by analytical modeling [15, 51], simulation [9, 12], or simply running the single-node implementation on one node of the target cluster. To figure out the effective bisection bandwidth (B_c) and the maximum communication overlap (O_c), we use Netgauge [32, 33] and Sandia MPI Micro-Benchmark Suite (SMB) [19] respectively. Netgauge measures the bisection bandwidth across a randomly-generated ring process topology to stress the different paths across the network. SMB measures the ability of the MPI library, run-time system, and network hardware to support independent progress of the non-blocking send-recv operations for different communication volumes (8-1M Bytes). This metric is platform-specific and represents the maximum overlap that can be achieved by the application using non-blocking communication.

Finally, the lower-bound on the parallel execution time is:

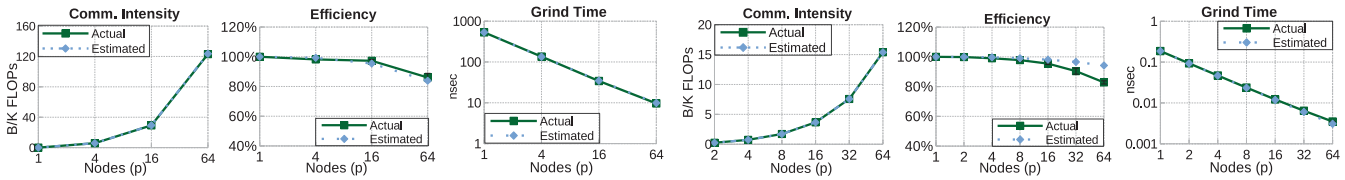
$$T_{p,l}(n,p) = \frac{T_1(n)}{p \times E_u(n,p)} \quad (5)$$

6 CASE STUDIES

We illustrate the capabilities of CommAnalyzer using two benchmarks and four real-world HPC applications. Table 1 presents the workloads considered in the case studies and the input data sets. We use strong scaling, where the total work is fixed on the different number of nodes, as it is the most challenging problem for the parallel efficiency prediction; in particular, at higher node counts, the average parallelism decreases, and the parallel/scheduling overhead and resource contention become the dominant efficiency bottlenecks. In addition, we explore the weak scaling problem for unstructured hydrodynamics code, which is the standard configuration for LULESH on HPC clusters [37]. The estimated communication and performance is evaluated in comparison with the actual MPI+OpenMP implementations. The MPI implementations utilize non-blocking communication to hide the communication latency, and use *MPI_THREAD_FUNNELED*, where the main thread calls the MPI communication APIs. The applications use double-precision floating point data types; however, the analysis works with any data type supported by LLVM.

6.1 Experimental Setup

As depicted in Figure 1, to estimate the communication at a large-scale input size n, the experiments leverage CommAnalyzer to analyze the sequential code of the target workloads with different problem sizes, where the largest problem size is at most $n/64$. Using CommAnalyzer's communication cost, scalability analysis tools can project the strong and weak scaling of the communication.



(a) MatMul (b) SPMV

Figure 6: Communication cost and performance of MatMul and SPMV benchmarks

Specifically, the experiments use Extra-P [11], a scalability analysis tool from Scalasca toolset [20], with the typical settings. Additional experiments with small-scale input data sets are available at [27].

Using the estimated communication cost, we project the efficiency and performance Roofline (Section 5) on the target HPC cluster. The performance is presented as the grind time, i.e., the execution time/iteration/data element, and it has the same estimation accuracy as the efficiency (see equation 5). The reported communication and performance is for the core application kernels and ignores the initialization, setup, and profiling code. The applications run for 100 iterations, and the experiments are repeated 5 times. The error bars show the 95% Confidence Interval.

6.1.1 Test Platform. The test platform is a Linux cluster consists of 196 nodes, and each node contains Intel Xeon processor E5-2683v4 (Broadwell) running at 2.10 GHz. The nodes are connected with an Intel OPA interconnect. The platform uses a batch queuing system that limits the number of nodes per user to 64 nodes. The test system runs CentOS Linux 7 distribution, and the applications are built using gcc 5.2 and OpenMPI 2.0. In the experiments, we launch one MPI process per node and limit the number of cores per process to 16, as using more cores increases the chance of high variance and reduced performance [5, 45].

To evaluate the efficiency Roofline model on the target cluster, we estimate the hardware parameters using the benchmarking approach detailed in Section 5. In particular, we use Netgauge-2.4.6 [32, 33] and SMB-1.0 [19]. Figure 7 shows the effective bisection bandwidth per node on the cluster. The average value of the communication overlap factor O_c on the test platform is 0.57, i.e., with enough computations to hide the data transfer, the application developer can overlap around 60% of the communication time on average. The application throughput per node $R(n)$ is estimated by running the single-node implementation on one node of the cluster.

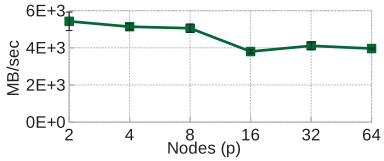


Figure 7: Effective bisection bandwidth per node

6.1.2 Analysis Overhead. As explained in Section 4.3, the time and memory overheads of CommAnalyzer are bounded by the VCG (Value Communication Graph) size, which depends on the inherent flow of information (communication) in the sequential code and can be quadratic in the worst case (if the code is fully connected). Table 2 shows the growth of the VCG size as a function of the problem size for the test applications and the size of the largest VCGs used in the experiments[3]. The results show that the VCG size grows much slower than the worst case even for dense linear algebra (MatMul). In most applications, the VCG size grows linearly with the problem size (i.e., each program value is connected to a limited number of neighboring values), while the VCG size of MatMul is $\approx O(n^{1.5})$.

Table 2: Value communication graph (VCG) size

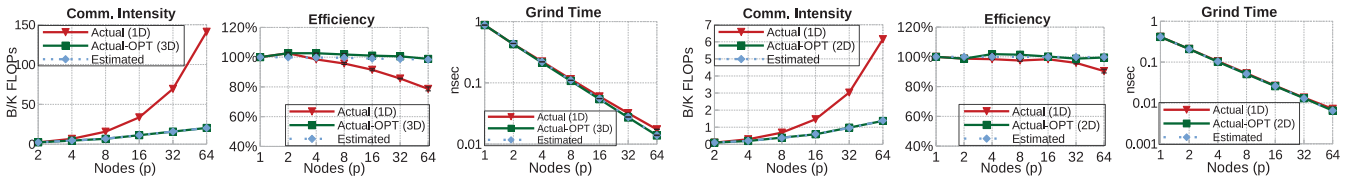
Application	Problem size n	VCG size (n)	Largest VCG size
MatMul	Matrix size	$O(4\ n^{1.5})$	8.01 GBs
SPMV	Sparse matrix size	$O(2\ n)$	0.5 GBs
MiniGhost	Grid size	$O(16\ n)$	4.1 GBs
Heat2D	Grid size	$O(12\ n)$	3 GBs
LULESH	Mesh size	$O(1568.2\ n)$	5.05 GBs
K-means	Data objects size	$O(2\ n)$	0.16 GBs

6.2 Benchmarks

First, we evaluate the accuracy of CommAnalyzer using two canonical regular and irregular workloads: MatMul and SPMV. MatMul is a traditional matrix multiplication. The MPI implementation of MatMul uses block-cyclic domain decomposition to distribute the matrices over the processes using 2D (square) process topology. SPMV is a sparse matrix vector multiplication using compressed sparse row (CSR) format. SPMV is an irregular application with indirect memory accesses through index arrays. To minimize the communication, the MPI implementation of SPMV uses 1D decomposition to distribute row blocks of the sparse matrix and chunks of the RHS (input) and LHS (output) vectors. Each process computes a part of the LHS 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 $(p-1)$ remote chunks of the RHS vector. Finally, the processes exchange the LHS vector using collective communication.

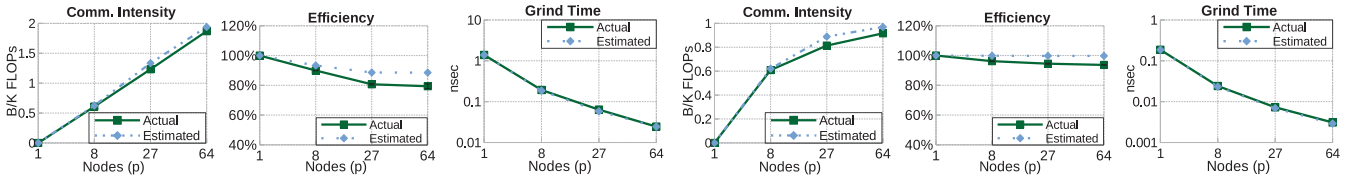
Figure 6 shows the estimated communication and performance in comparison with the actual MPI+OpenMP implementations for MatMul and SPMV. For the two benchmarks, the actual communication intensity is bounded by 98% of the estimated value. In SPMV, CommAnalyzer detected n output singleton values, where n is the vector size, corresponding to LHS. In addition, due to the density of the input matrix, CommAnalyzer identified n input singleton values which constitutes RHS. The actual MPI+OpenMP implementations achieves 99% and 96% of the efficiency upper-bound on average for MatMul and SPMV, respectively, and the maximum efficiency gap is 12%. At 64 nodes, SPMV attains 88% of the projected efficiency, as the execution time per iteration drops to few milliseconds and fixed overheads, such as the parallelization/scheduling overhead and communication setup, become the dominant efficiency bottlenecks.

[3]Detailed examples are available at https://github.com/vtsynergy/CommAnalyzer.



(a) MiniGhost (b) Heat2D

Figure 8: Communication cost and performance of MiniGhost and Heat2D



(a) Strong Scaling (b) Weak Scaling

Figure 9: Communication cost and performance of LULESH

6.3 MiniGhost and Heat2D

MiniGhost [6, 7] and Heat2D [44] 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 application for 3D 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 (PDEs) of heat diffusion in a homogeneous 2D space. The two applications use the iterative finite-difference method with explicit time-stepping scheme to solve the simulation equations. In particular, MiniGhost and Heat2D use 3D 7-point and 2D 5-point finite-difference stencils, respectively.

Typically, the distributed-memory implementation of structured grids applications uses domain decomposition 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 (2D faces and/or 1D 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).

Figures 8 shows the communication intensity and performance of MiniGhost and Heat2D. The two applications use 1D domain decomposition/process topology by default to reduce the message packing/unpacking overhead, and to reduce the programming/debugging effort. However, the results show that the projected efficiency Roofline is only attainable using 3D and 2D domain decomposition/process topologies for MiniGhost and Heat2D, respectively. In particular, when 1D domain decomposition is used, the efficiency gap can be as large as 20% with one order-of-magnitude higher communication intensity. When the MPI applications use multi-dimensional domain decomposition, the actual communication intensity reaches 97% of the estimated value on average. In some cases, the optimized MPI applications slightly exceed the projected parallel efficiency, due to the improved cache performance in comparison with the single-node implementations.

6.4 LULESH

LULESH [37] is a Lagrangian shock hydrodynamics simulation that represents more than 30% of the DoD and DoE workloads [42], and one of the five DARPA UHPC challenge problems. LULESH solves the Sedov blast wave problem [48] in 3D 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). In the Lagrangian hydrodynamics simulation, the mesh follows the motion of the elements in the space and time. 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 3D cube process topology/domain decomposition 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. 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: 3D nearest neighbor, 3D sweep, and collective broadcast and reduction [47].

Typically, LULESH uses weak-scaling on HPC clusters [37] because the single-node throughput significantly drops as the problem size decreases due to the parallelization overhead, e.g., additional data motion to handle race conditions. In addition, the message setup time is relatively large due to the packing/unpacking of 12 data fields with different memory-access strides. However, in the

experiment, we use both strong and weak scaling to show the efficiency gap in the presence of these overheads. Figures 9 describes the estimated and actual communication intensity and performance for LULESH. The communication intensity of the distributed memory implementation is within 95% of CommAnalyzer's prediction on average. In the weak-scaling problem, the actual implementation achieves 95% of the projected efficiency on average, and the maximum efficiency gap is 6%. The strong-scaling problem has lower efficiency (as discussed above), achieving 92% of the efficiency Roofline on average with a maximum gap of 11%.

6.5 K-means

K-means [39], one of the top 10 algorithms in data mining [57], is an unsupervised machine-learning application for data clustering that has been used in many fields, e.g., pattern recognition, bioinformatics, and statistics (outlier detection). For a given set of data objects in multi-dimensional features space, K-means iteratively finds k groups (clusters) of data objects based on feature similarities and generates a cluster membership label for every data object. A data object is considered to be in a cluster c, if it is closer to the centroid (mean) of c than that of any other clusters. In each iteration, K-mean computes the centroids (mean values) of the clusters based on their current data objects, and then generates a new membership label for each data object using similarity distance calculation.

The MPI implementation of K-means distributes the data objects evenly among the processes and uses global communication (reduction) to update the centroids of the clusters in every iteration. In particular, the computation cost of K-means is $O(n\ m\ k\ i)$, while its communication cost is $O(m\ k\ i)$, where n is the number of data objects, m is the feature vector size, k is the number of data clusters, and i is the number of iterations. Since n is generally much larger than m and k, the execution time of K-means is bounded by the computation time and achieves linear scaling on distributed-memory architectures.

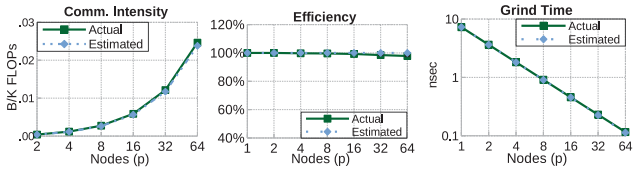


Figure 10: Communication and performance of K-means

Figure 10 shows the estimated communication and performance of K-means in comparison with the actual distributed-memory implementation. CommAnalyzer accurately detected the input/output singleton program values corresponding to the global (collective) communication required to update the centroids of the clusters. Overall, the actual communication intensity is bounded by 97% of the predicted value on average. Further, the distributed-memory implementation of K-means attains 99% of the projected efficiency Roofline on average, and the maximum efficiency gap is 3%.

7 DISCUSSION AND EXTENSIONS

CommAnalyzer estimates the communication of the distributed-memory parallel code that performs the same work (operations) as the given sequential code. Applying code optimizations that do not change how the output values are computed from the input values, such as loop tiling and data-layout transformations, does not affect the estimated communication cost. While the original sequential algorithm may not be the best one for parallel execution, the end user can still use CommAnalyzer to estimate the parallel communication cost of the algorithmic alternatives. Furthermore, CommAnalyzer assumes that the distributed-memory code adopts the SPMD execution model which is the dominant approach on HPC clusters [6, 18]. Other parallel execution models with data migration [10], i.e., the data moves according to the computations distribution, require a different communication estimation. One possible extension is to use a computation-centric analysis and to estimate the communication as the value-flow edges across compute instructions in different nodes.

Since CommAnalyzer instruments the sequential code to estimate its parallel communication, the problem size is limited by the available memory on a single node of the target cluster. Thanks to scalability analysis tools [11], the user can perform small-scale analysis experiments, which can be dealt with by CommAnalyzer, and project the strong and weak scaling of the communication. Even though we did not encounter any case where the scaling function could not be modeled by the existing scalability analysis tools, more sophisticated modeling techniques, such as machine learning, might be needed.

While CommAnalyzer adopts the sequential multilevel graph partitioning heuristic [38], other parallel and/or approximate graph partitioning heuristics can be used to reduce the analysis overhead. However, this might affect the quality of the VCG decomposition and the resulting accuracy of the communication estimation.

8 RELATED WORK

8.1 Communication Cost Estimation

Several approaches have been proposed for communication cost estimation from sequential code to enable the code generation for distributed-memory platforms [24, 30, 56]. The FORTRAN-D compiler [30] 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. Similarly Gupta et al. [24] adopt a compile-time, data-dependence analysis for estimating the communication of sequential code, which is only applicable for regular problems with array-based data structures. The SUIF compiler [56] solves the communication overestimation problem using the 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 model (owner-computes), Bondhugula [10] 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 dependencies. 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 parallel execution models.

8.2 Performance Prediction on HPC Clusters

Analytical Modeling. Analytical performance modeling provides useful insights into the application performance by modeling the interactions between the HPC platform and the application; however, it requires tedious manual analysis of the target applications and architectures. The LogP model family [3, 16] contains several *high-level* analytical models that predict the communication time and performance of MPI applications using a few parameters which abstract the application and hardware characteristics. ASPEN [51] is a modeling language for describing formal application/machine models to explore the algorithm and architecture design options. PALM [53] simplifies the performance modeling process by providing a modeling language to support the generation of analytical models from the annotated MPI code. Snavely et al. [50] 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.

Simulation. Distributed-memory simulators, such as LogGOPSim [34] and DIMEMAS [46], incorporate detailed network and architecture models to estimate the communication time and performance. However, they require the MPI parallel implementation to profile the communication operations and generate the application traces. MUSA [22] adopts a multi-level simulation approach with different levels of hardware details, simulation cost, and simulation accuracy. In addition, it identifies and simulates the representative application phases to reduce the overall simulation time.

Scalability Analysis. The scalability analysis tools [5, 11, 14, 59] project the performance of a given MPI implementation at massive scale based on small-scale experiments. Typically, these tools extract the communication, computation, and/or memory traces of the MPI application using dynamic instrumentation and profiling. 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 [49] 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 distributed-memory implementation of the original applications.

9 CONCLUSION

This work presents CommAnalyzer, a novel approach to predict the communication cost and scalability of the sequential code when executed on multiple compute nodes using the SPMD execution model. We implemented CommAnalyzer in the LLVM compiler framework and used novel dynamic program analyses and graph partitioning algorithms to estimate the minimum communication and maximum efficiency/performance on HPC clusters. The case studies, with two benchmarks and four real-world applications, demonstrate that CommAnalyzer predicts the communication intensity of sequential applications with more than 95% prediction accuracy on average. Moreover, the optimized MPI+X implementations achieve more than 92% of the projected efficiency upper-bound, while the maximum efficiency gap is 12%. The experiments represent the efficacy of CommAnalyzer for regular, irregular, and unstructured problems that have different communication patterns such as 2D/3D nearest neighbor, 3D sweep/wavefront, 2D broadcast, and collective broadcast/reduction communication.

There remain many opportunities to expand on the proposed tool. Integrating CommAnalyzer with existing scalability analysis tools enables the estimation of the optimal number of the required compute nodes for each application to achieve high HPC system utilization. Furthermore, CommAnalyzer is a perfect candidate to drive a communication-aware workload distribution scheme to efficiently utilize the available compute resources across multiple HPC nodes [26, 28], laying the foundation for addressing the communication challenges of Exascale computing.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Matei Ripeanu and the anonymous reviewers for their constructive comments and feedback. We also thank Alexandru Calotoiu of Laboratory for Parallel Programming at TU Darmstadt for his help with the Extra-P tool, and Mark Gardner, Vignesh Adhinarayanan, and Paul Sathre of the Synergy Lab at Virginia Tech for valuable discussions. This work was supported in part by NSF CAREER Award (1750503) and the Synergistic Environments for Experimental Computing (SEEC) Center via the Institute for Critical Technology and Applied Science (ICTAS), an institute dedicated to transformative, interdisciplinary research for a sustainable future. We acknowledge Advanced Research Computing (ARC) at Virginia Tech for the computational resources.

REFERENCES

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[2] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '90)*. 246–256.

[3] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. 1995. LogGP: Incorporating Long Messages into the LogP Model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA '95)*. 95–105.

[4] Amir Bahmani and Frank Mueller. 2014. Scalable performance analysis of exascale mpi programs through signature-based clustering algorithms. In *Proceedings of the ACM international conference on Supercomputing (ICS '14)*. 155–164.

[5] Bradley J Barnes, Barry Rountree, David K Lowenthal, Jaxk Reeves, Bronis De Supinski, and Martin Schulz. 2008. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing*. 368–377.

[6] Richard F Barrett, Simon D Hammond, Courtenay T Vaughan, Douglas W Doerfler, Michael A Heroux, et al. 2012. Navigating an Evolutionary Fast Path to Exascale. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC '12)*. 355–365.

[7] Richard F Barrett, Courtenay T Vaughan, and Michael A Heroux. 2011. *MiniGhost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing*. Technical Report.

[8] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, et al. 2008. *Exascale computing study: Technology challenges in achieving exascale systems*. Technical Report.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, et al. 2011. The gem5 simulator.

ACM SIGARCH Computer Architecture News 39, 2 (2011), 1–7.

[10] Uday Bondhugula. 2013. Compiling affine loop nests for distributed-memory parallel architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13)*. 33.

[11] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. 2013. 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 (SC '13)*. 45.

[12] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. 52.

[13] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. 1999. *Introduction to UPC and language specification.* Technical Report.

[14] Marc Casas, Rosa Badia, and Jesús Labarta. 2008. Automatic analysis of speedup of MPI applications. In *Proceedings of the International Conference on Supercomputing (ICS '08)*. 349–358.

[15] Jee Choi, Marat Dukhan, Xing Liu, and Richard Vuduc. 2014. Algorithmic time, energy, and power on candidate HPC compute building blocks. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '14)*. 447–457.

[16] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, et al. 1993. 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)*. 1–12.

[17] Roxana E Diaconescu and Hans P Zima. 2007. An approach to data distributions in Chapel. *International Journal of High Performance Computing Applications* 21, 3 (2007), 313–335.

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

[19] Douglas Doerfler and Ryan Grant. 2018. Sandia MPI Micro-Benchmark Suite. http://www.cs.sandia.gov/smb/index.html. Accessed: 2018.05.11.

[20] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. 2010. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 702–719.

[21] Ananth Grama, Anshul Gupta, and Vipin Kumar. 1993. 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*.

[22] Thomas Grass, César Allande, Adrià Armejach, Alejandro Rico, Eduard Ayguadé, Jesus Labarta, Mateo Valero, et al. 2016. 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 (SC '16)*. 45.

[23] Anshul Gupta and Vipin Kumar. 1993. Performance properties of large scale parallel systems. *J. Parallel and Distrib. Comput.* 19, 3 (1993), 234–244.

[24] Manish Gupta and Prithviraj Banerjee. 1992. Compile-time estimation of communication costs on multicomputers. In *Proceedings of the International Parallel Processing Symposium*. 470–475.

[25] Ahmed E Helal, Amr M Bayoumi, and Yasser Y Hanafy. 2015. Parallel circuit simulation using the direct method on a heterogeneous cloud. In *Proceedings of the 52nd Annual Design Automation Conference (DAC '15)*. 186.

[26] Ahmed E. Helal, Wu-chun Feng, Changhee Jung, and Yasser Y. Hanafy. 2017. AutoMatch: An Automated Framework for Relative Performance Estimation and Workload Distribution on Heterogeneous HPC Systems. In *2017 IEEE International Symposium on Workload Characterization (IISWC '17)*. 32–42.

[27] Ahmed E. Helal, Wu-chun Feng, Changhee Jung, and Yasser Y. Hanafy. 2017. *CommAnalyzer: Automated Estimation of Communication Cost on HPC Clusters Using Sequential Code.* Technical Report.

[28] Ahmed E. Helal, Paul Sathre, and Wu-chun Feng. 2016. 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)*. 11:1–11:11.

[29] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, and et al. 2009. *Improving performance via mini-applications.* Technical Report.

[30] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. 1992. Compiling Fortran D for MIMD distributed-memory machines. *Commun. ACM* 35, 8 (1992), 66–80.

[31] Victoria Hodge and Jim Austin. 2004. A survey of outlier detection methodologies. *Artificial intelligence review* 22, 2 (2004), 85–126.

[32] Torsten Hoefler, Torsten Mehlan, Andrew Lumsdaine, and Wolfgang Rehm. 2007. Netgauge: A Network Performance Measurement Framework. In *Proceedings of the International Conference on High Performance Computing and Communications (HPCC'07)*. 659–671.

[33] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2008. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *Proceedings of the IEEE International Conference on Cluster Computing*. 116–125.

[34] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. LogGOPSim: Simulating Large-scale Applications in the LogGOPS Model. In *Proceedings of the International Symposium on High Performance Distributed Computing*. 597–604.

[35] RD Hornung, JA Keasler, and MB Gokhale. 2011. *Hydrodynamics challenge problem.* Technical Report. Lawrence Livermore National Laboratory (LLNL).

[36] Nikhil Jain, Abhinav Bhatele, Michael P Robson, Todd Gamblin, and Laxmikant V Kale. 2013. Predicting application performance using supervised learning on communication features. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13)*. 95.

[37] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L Chamberlain, Jonathan Cohen, Zachary Devito, et al. 2013. Exploring traditional and emerging parallel programming models using a proxy application. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. 919–932.

[38] George Karypis and Vipin Kumar. 1998. Multilevel Algorithms for Multi-constraint Graph Partitioning. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC '98)*. 1–13.

[39] Wei keng Liao. 2018. The software package of parallel k-means. http://users.eecs.northwestern.edu/wk-liao/Kmeans/index.html. Accessed: 2018.05.11.

[40] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization (CGO '04)*. 75.

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

[42] DoD High Performance Computing Modernization Program Office. 2010. *Department of Defense High Performance Computing Modernization Program FY 2010 Requirements Analysis Report.* Technical Report. DTIC Document.

[43] OpenMP, ARB. 2013. OpenMP 4.0 specification.

[44] Necati Ozisik. 1994. *Finite difference methods in heat transfer.* CRC press.

[45] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. 2003. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC '03)*. ACM, 55–.

[46] German Rodriguez, Rosa Badia, and Jesús Labarta. 2004. Generation of simple analytical models for message passing applications. In *Euro-Par 2004 Parallel Processing*. 183–188.

[47] Philip C Roth, Jeremy S Meredith, and Jeffrey S Vetter. 2015. Automated Characterization of Parallel Application Communication Patterns. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*. 73–84.

[48] LI Sedov. 1959. Similarity and Dimensional Methods in Mechanics, Acad. *Press, New York* (1959).

[49] Sameer S Shende and Allen D Malony. 2006. The TAU parallel performance system. *The International Journal of High Performance Computing Applications* 20, 2 (2006), 287–311.

[50] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. 2002. A framework for performance modeling and prediction. In *Proceedings of the ACM/IEEE conference on Supercomputing*. 21–21.

[51] Kyle L. Spafford and Jeffrey S. Vetter. 2012. Aspen: A Domain Specific Language for Performance Modeling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*. 84:1–84:11.

[52] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin Slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. 112–122.

[53] Nathan R. Tallent and Adolfy Hoisie. 2014. Palm: Easing the Burden of Analytical Performance Modeling. In *Proceedings of the ACM International Conference on Supercomputing (ICS '14)*. 221–230.

[54] Heidi K Thornquist, Eric R Keiter, Robert J Hoekstra, David M Day, and Erik G Boman. 2009. 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*. 410–417.

[55] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[56] Robert P Wilson, Robert S French, Christopher S Wilson, Saman P Amarasinghe, Jennifer M Anderson, Steve WK Tjiang, Shih-Wei Liao, et al. 1994. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices* 29, 12 (1994), 31–37.

[57] Xindong Wu, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, Angus Ng, et al. 2008. Top 10 algorithms in data mining. *Knowledge and information systems* 14, 1 (2008), 1–37.

[58] Xing Wu, Frank Mueller, and Scott Pakin. 2011. Automatic generation of executable communication specifications from parallel applications. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. 12–21.

[59] Jidong Zhai, Wenguang Chen, and Weimin Zheng. 2010. Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. In *In Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 305–314.