Network-Theoretic Classification of Parallel Computation Patterns

Sean Whalen
Berkeley Lab and
University of California, Davis
shwhalen@ucdavis.edu

Sean Peisert
Berkeley Lab and
University of California, Davis
sppeisert@lbl.gov

Matt Bishop
University of California, Davis
bishop@cs.ucdavis.edu

ABSTRACT

Parallel computation in a high performance computing environment can be characterized by the distributed memory access patterns of the underlying algorithm. During execution, networks of compute nodes exchange messages that indirectly exhibit these access patterns. Thus, identifying the algorithm underlying these observable messages is the problem of latent class analysis over information flows in a computational network. Towards this end, our work applies methods from graph and network theory to classify parallel computations solely from network communication patterns. We also introduce an approximate pattern matching algorithm using statistical hypothesis testing and compare these approaches using massive datasets collected at Lawrence Berkeley National Laboratory.

Categories and Subject Descriptors

I.5.1 [**Pattern Recognition**]: Models—*statistical, structural*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*

Keywords

communication patterns, network theory, hypothesis testing

1. INTRODUCTION

As the field of high performance computing (HPC) plans for the frontier of exascale performance, multi-core CPUs are often combined with accelerators such as graphics processing units (GPUs) and field programmable gate arrays to form heterogeneous environments that increase performance under both power and transistor constraints [1]. While architectural evolution is one cornerstone of this frontier, understanding the characteristics of distributed computation is essential for improving the efficiency and scalability of software in HPC environments [2].

Towards this end we apply methods from graph theory, network theory, and hypothesis testing to classify distributed parallel computations based on their characteristic patterns of communication. More generally, classification is the problem of structural pattern recognition applied to unknown parallel computations that reveal themselves indirectly via messages exchanged by a network of compute nodes. Identifying the underlying algorithm is thus a type of latent class analysis where a "hidden" algorithm must be identified only from observable information flows. This task is non-trivial: different algorithms can express similar patterns, and different implementations can express different patterns.

This work was initially motivated by the need for anomaly detection in HPC environments given the recent commodification of cloud-based encryption and password cracking services.[1] However, it has broader applications in heterogeneous computing environments. Consider an algorithm implemented on a general purpose CPU and later ported to a GPU. Have algorithms with similar patterns on the CPU been successfully ported to accelerators in the past? How close is the new pattern to those of similar algorithms on the GPU? Can we distinguish CPU and accelerator implementations based solely on information flows? The classification of communication patterns is a step towards answering such questions and may help identify algorithms suitable for particular architectures or dynamic replacement of suboptimal implementations [3, 4].

In this paper, we first describe how communication patterns are defined and dynamically captured from running applications and the relation of these patterns to abstract computational classes called *dwarfs*. Methods from graph and network theory are then applied to the classification problem and their shortcomings discussed. Finally, we introduce a new algorithm for efficient, approximate matching of communication patterns and suggest directions for developing metric-based pattern classifiers.

2. BACKGROUND

2.1 Message Passing Interface

Message Passing Interface (MPI) is a communications protocol standard used by many parallel programs to exchange data using a distributed memory model. There are several implementations such as OpenMPI and MPICH, each based on the idea of logical processors with unique labels called *ranks* communicating in groups called *communicators*. MPI

[1]An unnamed website brute forces WPA-PSK or ZIP passwords in 20 minutes for $17 using a cloud computing infrastructure, and Amazon's EC2 GPU instances have been used to brute force SHA-1 hashes.

```
1  int main(int argc, char **argv) {
2      MPI_Init(&argc, &argv);
3      long terms_total = atol(argv[1]);
4
5      int rank, size;
6      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7      MPI_Comm_size(MPI_COMM_WORLD, &size);
8
9      double pi;
10     calculate_pi(terms_total * rank / size, terms_total * (rank + 1) / size, &pi);
11
12     if(rank == 0) {
13         double data;
14         MPI_Status status;
15
16         for(int i = 1; i < size; i++) {
17             MPI_Recv(&data, 1, MPI_DOUBLE, i, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
18             pi += data;
19         }
20
21         printf("%.18f", pi);
22     }
23     else {
24         MPI_Send(&pi, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
25     }
26
27     MPI_Finalize();
28 }
```

Figure 1: Source code for computing the decimal expansion of π using the Message Passing Interface parallel programming standard. The code is left unoptimized for the benefit of those not familiar with MPI.

programs have an initialization phase where each processor joins a communicator and is assigned a rank, and a finalization phase to gracefully terminate after computation.

Consider the MPI-enabled C program for computing π to some number of digits shown in Figure 1.[2] Lines 2, 6, and 7 perform MPI initialization. The desired number of terms in the decimal expansion is read from the command line into `terms_total`. Each processor then calls `calculate_pi` and tells the function which one of `size` chunks to compute based on its rank. After computing its chunk, every rank sends its result to rank 0 on line 24. Rank 0 collects these results on line 17 and adds them to the final output, displayed to the desired precision on line 21. This is an example of a simple communication pattern: all ranks call `MPI_Send` with destination rank 0.

Sophisticated applications have richly structured communication patterns between many ranks in the communicator and we use these patterns to classify parallel computations. However, not all communication attributes (*features*) may be helpful in classification—some may be redundant or even detrimental to training models that generalize to new data. We later describe how these features are selected, but first detail how they are captured from active MPI applications.

2.2 Communication Logging

The *Integrated Performance Monitoring* (IPM) library [5] provides low-overhead performance and resource profiling for parallel programs. It logs features of MPI function calls to a file such as the call name, the ranks involved, the number of bytes sent, and optional hardware counters including the number of integer and floating point operations. The library is enabled with a compile-time flag, usually provided to the MPICC compiler wrapper.

Consider the following IPM log entry:

<hent call="MPI_Isend" bytes="599136" orank="1" count="26" />

These entries become rows in a two dimensional feature matrix where rows represent individual calls and columns represent the features of each call. Functions names are mapped to unique integers so the contents of the feature matrix are purely numerical. The above entry then becomes:

$$\begin{pmatrix}\text{int(MPI\_Isend)} & 599136 & 1 & 26\end{pmatrix}$$

The result is a matrix of features for each parallel program we wish to classify, and from this we construct a directed graph. The task at hand, then, is to differentiate patterns of computation by assessing the pattern of information flow between nodes in the graph, and what captured features are important to this analysis.

2.3 Computational Dwarfs

A computational dwarf is "a pattern of communication and computation common across a set of applications" [6]. Each dwarf is an equivalence class of computation and is invariant to the programming language or numerical methods used for implementation. The common use of shared

[2]Adapted from code by David Letscher at St. Louis University.

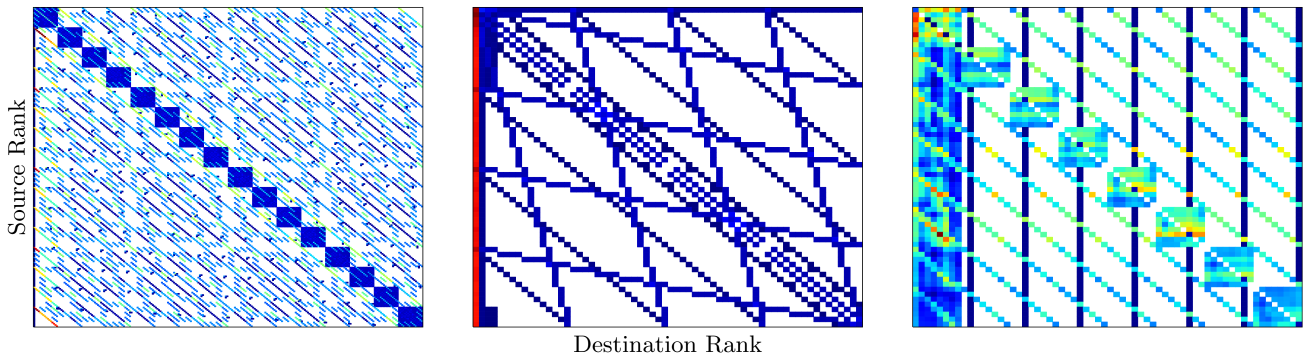


Figure 2: Adjacency matrices for individual runs of performance benchmark MADBENCH (256 nodes), atmospheric dynamics simulator FVCAM (64 nodes), and linear equation solver SUPERLU (64 nodes). The number of bytes sent between ranks is linearly mapped from dark blue (lowest) to red (highest), with white indicating an absence of communication.

libraries such as BLAS and LAPACK provides some evidence of these equivalence classes, though dwarfs imply a level of algorithmic equivalence beyond code reuse.

Colella et al. identified seven dwarfs in HPC applications [7]: dense linear algebra, sparse linear algebra, spectral methods, n-body methods, structured grids, unstructured grids, and monte carlo methods. Asanovich et al. asked if these seven also captured patterns from areas outside of HPC [6]. They found six additional dwarfs were needed to capture the distinct patterns of computation found in areas such as machine learning, computer graphics, and databases.

The distinct adjacency matrices of dwarfs provide a first step towards classification. Consider a three node cluster where rank 0 sends messages to ranks 1 and 2, rank 1 sends a message to rank 2, and ranks 1 and 2 send messages back to 0. This leads to the following matrix representation:

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

Adjacency matrices are plotted as a grid where the axes are rank numbers and filled pixels denote communicating ranks. The matrices for single runs of three parallel programs are shown in Figure 2 and exhibit distinct patterns. Another layer of structure can be seen by extending the adjacency matrix into a third dimension, mapping an additional communication feature onto the z-axis. Two such mappings are shown in Figure 3.

Communication patterns are strongly tied to distributed memory access within a parallel program. To see this, examine the diagonal of Figure 2's center panel and note the communication between a rank and its immediate neighbors. Such a pattern is a signature of finite difference equations and is found across many HPC applications. Another type of equation will have a different signature unless its pattern of distributed memory access is similar.

The structure seen in Figures 2 and 3 is typical of MPI applications and suggests that classification is possible. By the same argument, however, distinguishing applications within the same dwarf class may be difficult due to their topological similarity. Complicating matters, the same program may alter its communications given different parameter values, datasets, or communicator sizes (see Figure 5). As a result, we must look beyond the matrix representations of these patterns.

3. METHODOLOGY

3.1 Node Distributions

We treat a communication pattern as a directed graph with ranks as nodes and MPI calls as edges[3] and measure various statistical properties of this graph. The first of these, the *node degree distribution*, counts the total number of nodes having a particular number of edges (*degree*). This analysis is restricted to the out-degree distribution measuring only outbound edges. In our previous example, two nodes have degree 2 and a single node has degree 1. The degree distribution for two individual runs of the MADBENCH performance benchmark are shown in panels (a) and (c) of Figure 4.

Node degree distributions are a summary statistic over the adjacency matrix and the types of messages exchanged, reflecting the layered per-call adjacency matrices in the left panel of Figure 3. Though offering additional structural insight, node degree distributions summarize a single aspect of the graph that may fail to distinguish different patterns. In these cases, the notion of *centrality* can be helpful.

Centrality measures the importance of a node in the graph, and this importance can be defined in several ways. We examine the *betweenness centrality* (C_B), measuring the percent of shortest paths passing through a node v in an undirected graph [8]:

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where $\sigma_{st}(v)$ is the number of shortest paths between nodes s and t that pass through v. This number is normalized by the total number of shortest paths between s and t. The C_B of v, then, is the sum of these normalized shortest path counts over all node pairs not containing v.

Intuitively, nodes acting as coordinators of computation such as rank 0 have high C_B. As an artifact of IPM, broad-

[3]Each logged call is given a unique edge, but later graphs are restricted to one edge per message type to reduce runtime and overfitting. For example: if rank 1 transmits three `MPI_Recv` messages to rank 2, it will result in three edges here but a single edge later.

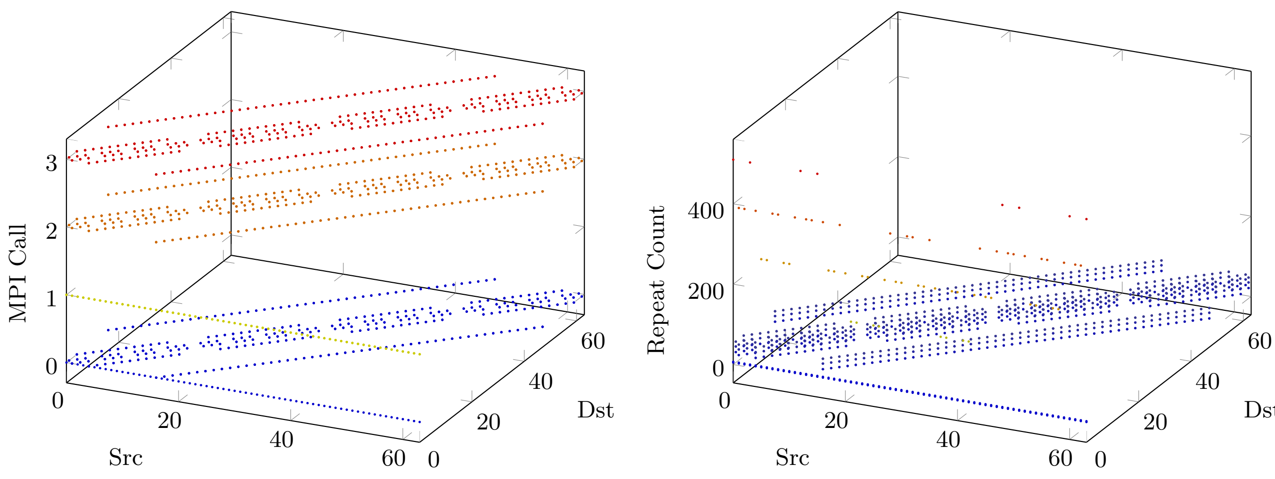


Figure 3: Adjacency matrix of general relativity simulator CACTUS augmented by MPI call (left) and message size (right).

cast messages also have high centrality. This can be seen in panels (b) and (d) of Figure 4. Note that despite similar degree distributions, the second run has very different centrality distributions.

Relying solely on centrality to classify the computation would have resulted in a false negative (incorrectly labeling patterns as different algorithms). The degree distribution works for this example but will result in a false positive (incorrectly labeling patterns as the same) for others. Thus, multiple such measures are important for differentiating parallel computations.

However, these statistics are based solely on topological properties of the computational network and do not incorporate other attributes of information flow. In addition they are potentially sensitive to the number of compute nodes. It is vital that any classification operate independently of the communicator size, whether the computation is performed with 32, 64, 128, or more nodes.

3.2 Graph Isomorphisms

In our setting, two parallel computations using the same algorithm are often *isomorphic*: given the set of vertices $V(G)$ and edges $E(G)$ for some graph G, graph isomorphism is a bijection between two graphs G and H:

$$f : V(G) \rightarrow V(H)$$

This mapping preserves the edge structure of the graphs: $uv \in E(G)$ if and only if $f(u)f(v) \in E(H)$ [9]. Isomorphic communication patterns imply that messages are passed between the same nodes in each graph regardless of the assigned rank number, akin to scrambling the columns of the adjacency matrix.

As two computations performed on different numbers of nodes cannot be isomorphic, we are instead interested in the related problem of *subgraph isomorphism*. Two graphs are subgraph isomorphic if some subgraph of G is isomorphic to H: for example, if some subset of the communication graph for atmospheric dynamics simulator FVCAM on 256 nodes is isomorphic to the same program run with 128 nodes.

Graph isomorphism has not been proven NP-complete or a member of P, while subgraph isomorphism is NP-complete via reduction to the maximum clique problem. Runtime complexity is of serious concern as HPC networks often contain hundreds of nodes. Our investigations focus on Ullman's subgraph isomorphism algorithm [10], the VF2 algorithm [11], and the NetworkX implementation of VF2 [12]. The Ullman and VF2 algorithms are implemented in the VFLib library[4] and have worst-case time complexity $\mathcal{O}(N!N^2)$ and $\mathcal{O}(N!N)$, respectively.

IPM logs are converted to directed graphs with MPI calls encoded as edge attributes. Such *attributed relational graphs* (ARGs) [13] prune the state space of the isomorphism test and reduce false positives. The results of pairwise subgraph isomorphism tests are shown in the left panel of Figure 6 and full details of the datasets are given in Section 3.4. True positives are shown in black and false negatives in grey. Blocks represent graphs constructed from different runs of the same program and thus should be classified the same.

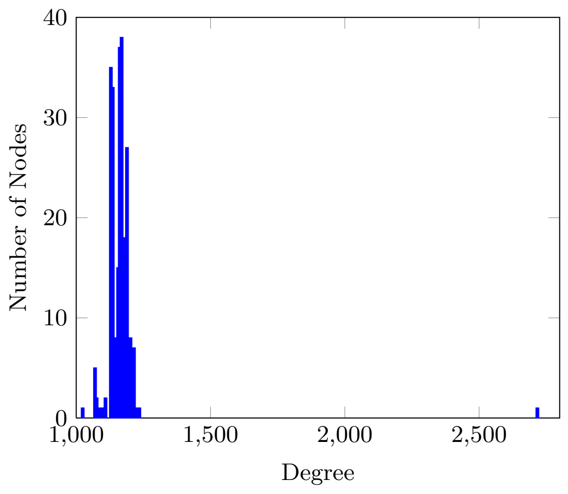
These blocks illustrate the primary problem with isomorphism testing: exact matching requirements result in false negatives for all data-dependent topologies (see Figure 5). Ideally, such topologies will be classified the same if differences are within some bounded variance. To address this, the next section introduces an approximate matching algorithm based on a series of statistical hypothesis tests and is followed by a comparison of both approaches.

3.3 Hypothesis Testing

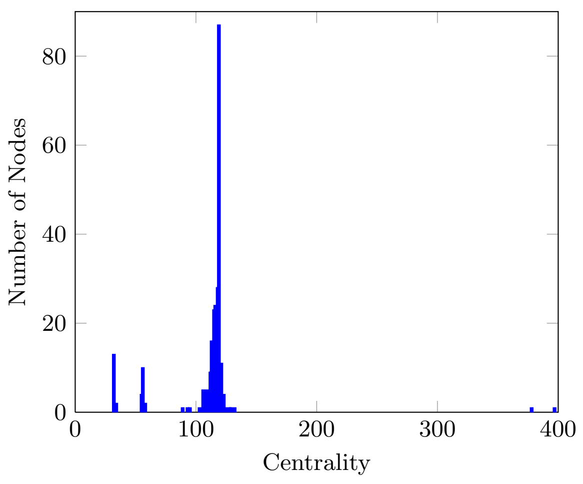
A graph isomorphism requires exact node correspondence between two graphs. This requirement results in many false negatives when applied to communication patterns whose statistics can vary with architecture, communicator size, parameters, and datasets. Current approximate graph matching methods such as graph edit distance [13] are often more computationally expensive than exact graph matching. Instead, we present a matching algorithm based on statistical hypothesis testing. First we review the relevant concepts and notation then discuss our approach in these terms.

A hypothesis test is a statistical method to determine whether a null hypothesis H_0 or alternative hypothesis H_a best explain some data [14]. The null hypothesis commonly

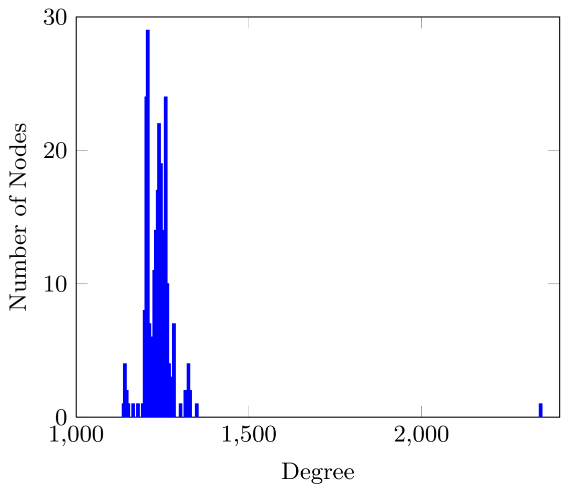
[4]`http://www.cs.sunysb.edu/~algorith/implement/vflib/implement.shtml`



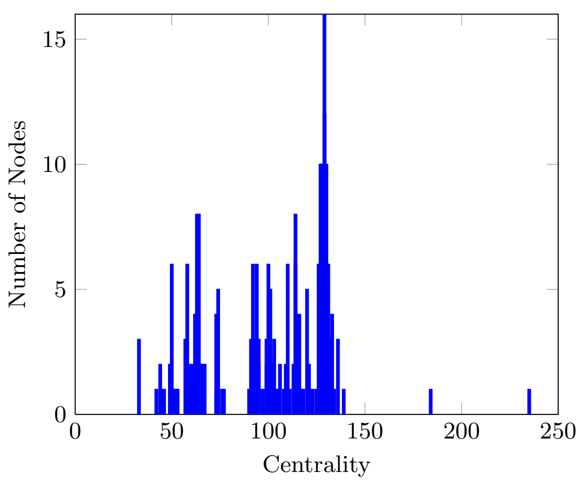
(a) Node degree distribution, MADBENCH run 1



(b) Centrality distribution, MADBENCH run 1



(c) Node degree distribution, MADBENCH run 2



(d) Centrality distribution, MADBENCH run 2

Figure 4: Node degree and betweenness centrality distribution for two runs of performance benchmark MADBENCH. Despite similar node degree distributions these runs exhibit different centrality distributions.

theorizes the data is a result of chance and is accepted or rejected at a significance level α using some statistical test. If rejected, H_a is accepted as true with α probability of a type-I error (false positive).

A type of hypothesis test called a *goodness-of-fit* test can be used to determine the equality of probability distributions. We use the two-sample Kolmogorov-Smirnov (KS) test [15], first computing the *D-statistic* for two empirical cumulative distribution functions:

$$D_{m,n} = \max_x |\hat{S}_m(x) - \hat{S}_n(x)|$$

where m and n are the total event counts of their respective distributions. We then compute the probability that differences in the distributions are due to chance and reject if this value is less than our threshold α:

$$P(D_{m,n} \geq D_O | H_0) < \alpha$$

for the observed value D_O [14]. Though defined theoretically for continuous distributions, a modified KS test can be used with discrete distributions [16].

To perform a KS test, the counts of MPI messages sent by each rank are normalized to form a probability distribution and the cumulative sum is taken to produce $\hat{S}$. For example: summed over all destination ranks, source rank 1 of program A may transmit 15 `MPI_Send`, 20 `MPI_Recv`, and 5 `MPI_Barrier` messages. The probability mass function is then 37.5%, 50%, and 12.5% respectively for these calls and 0% for all others. If source rank 1 of program B transmits 18 `MPI_Send`, 19 `MPI_Recv`, and 4 `MPI_Barrier` messages, the probability mass function is 43.9%, 46.3%, and 9.7%. Taking their respective cumulative sums to obtain $\hat{S}_{40}$ and $\hat{S}_{41}$, the KS test determines that these distributions are not significantly different at the $\alpha = 0.01$ level. If the probability mass functions remained the same but instead 2000 calls were logged, the test would determine the distributions are not the same at the $\alpha = 0.01$ level since there is far more data and the differences are less likely due to chance.

To determine if two communication patterns are generated by the same program, a KS test is applied to identi-

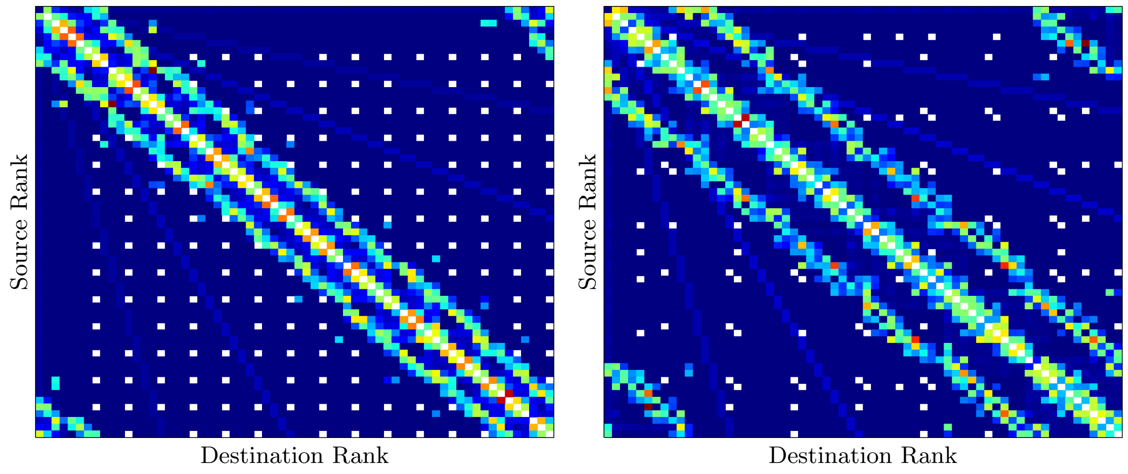


Figure 5: Data dependent topology demonstrated by molecular dynamics simulator NAMD under different molecular arrangements. The number of bytes sent between ranks is linearly mapped from dark blue (lowest) to red (highest), with white indicating an absence of communication.

cal ranks in the communicators. For example: rank 1 of program A is compared to rank 1 of program B, rank 2 compared to rank 2, and so on. For communicators of different size, the comparisons are performed between ranks present in the smallest communicator only. If more than some threshold of ranks are equivalent at significance level α, the programs are deemed equivalent. Both parameters provide an adjustable tolerance to topological differences. We found half the size of the smallest communicator to be an effective threshold.

3.4 Comparison

Hypothesis testing results are shown in the right panel of Figure 6 and both approaches are summarized in the table below. Our 31 gigabyte dataset consists of 202 IPM logs generated by CACTUS (astrophysics), FVCAM (atmospheric dynamics), GTC (particle physics), HYPERCLAW (gas dynamics), LBMHD (plasma physics), MADBENCH (benchmark), MHD (plasma physics), NAMD (molecular dynamics), PARATEC (materials science), PF_2 (plasma physics), PMEMD (molecular dynamics), PSTG3R (atomic physics), SUPERLU (linear equation solver), and SU(3) (lattice gauge theory). Programs are logged with different combinations of 32, 64, 128, and 256 ranks. A *true positive* denotes matching patterns generated by different runs of the same program; a *false negative* occurs when these patterns do not match. Similarly, a *true negative* occurs when patterns generated by different programs do not match; a *false positive* occurs when they do. A perfect classification algorithm will have only true positives and true negatives.

	Test	
	Subgraph Isomorphism	KS
True Positives	922	2930
False Negatives	2542	534
True Negatives	14492	14398
False Positives	0	94
Total Runtime	17m40s	23s

Note the small increase in false positives for the KS test due to approximate matching. We consider this an acceptable tradeoff for the large reduction in false negatives and runtime compared to subgraph isomorphism testing.

4. RELATED WORK

IPM logs have previously been used to study the performance of MPI applications. Fürlinger et al. [17] provide a general introduction to the IPM package and discuss several concepts related to this work including visualization of adjacency matrices and examining the distribution of MPI calls for an entire process. Shalf et al. [18] perform similar analysis to evaluate the communication requirements of parallel programs for improving processor interconnect designs. The adjacency matrices of several NAS parallel benchmarks, augmented by number of messages and message size, are presented by Riesen [19].

Ma et al. [20] introduce a communication correlation coefficient to characterize the similarity of parallel programs using several metrics. The first compares the average transmission rate, message size, and unique neighbor count for each rank, while the second computes the maximum common subgraph. Their evaluation was limited to 4 programs in the NAS parallel benchmark.

In addition to graph and network theory, we have previously used machine learning to classify parallel computation [21, 22] and discuss related machine learning efforts elsewhere. This approach achieves greater accuracy than those found in Section 3.3 but requires substantially more computation and care to prevent overfit models.

Other parallel programming standards such as OpenMP are based on a shared, as opposed to distributed, memory model. In an effort to increase the portability of parallel software, recent work uses compiler techniques to translate OpenMP into MPI source code [23, 24], and our approach should apply when such techniques are used. While we focus on latent analysis using only runtime communications, source code translation has also been used to replace inefficient computations [3, 4]. The classification of communication patterns thus has strong ties to compilers, static analysis, and code optimization.

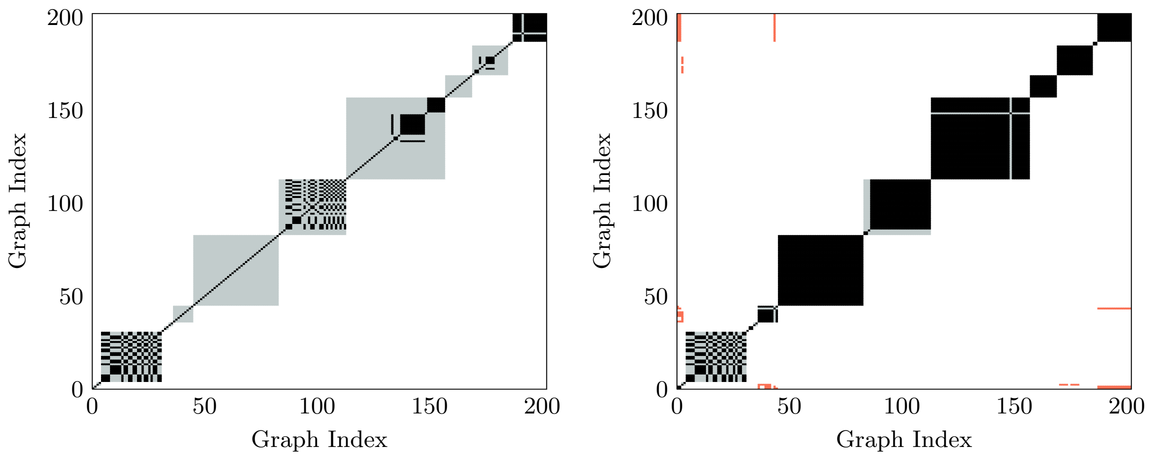


Figure 6: Pairwise comparison of 202 communication patterns using subgraph isomorphism testing (left) and hypothesis testing (right). Comparisons are symmetric and reflect along the diagonal. Blocks represent graphs constructed from different runs of the same program and thus should be classified the same. True positives, false negatives, and false positives are shown in black, grey, and red, respectively.

5. CONCLUSION

This work applies methods from graph and network theory to identify the latent class of a parallel computation from the observable information passed between nodes in a computational network: given logs of MPI messages from an unknown program, the task is to infer the program most likely to have generated those logs. Our original motivation was the detection of anomalous behavior on HPC clusters, though we present our work in a general context and suggest applications to heterogeneous computing are possible.

As initially postulated by work on computational dwarfs [6, 7], communication patterns tend to be highly structured and reflect the distributed memory access patterns of the underlying algorithm. When dealing with algorithm *implementations*, however, many other factors affect the communication patterns of theoretical algorithms. Different implementations of the same algorithm, shared libraries, compiler optimizations, architecture differences, software flaws, debug flags, and numerous MPI implementations all make this task more difficult. Further, some parallel programs have data-dependent communication topologies, varying both slightly (see Figure 5) and greatly as with multi-use ("swiss-army") libraries or interpreters such as Matlab.

Using gigabytes of data covering over a dozen parallel programs, we constructed directed communication graphs and found network-theoretic measures such as node degree and centrality distributions capture insufficient information to distinguish parallel computation. Adjacency matrices serve to visually differentiate communication patterns but cannot group data-dependent topologies. Towards this end we examined graph and subgraph isomorphisms for testing equivalency with the latter allowing comparison between communicators of different size. While isomorphism tests work for small or sparse graphs, we found many topologies exhibit the worst-case exponential time complexity of these algorithms. In addition, isomorphism tests are not sufficiently tolerant of data-dependent computation.

To address this issue we developed an algorithm that performs statistical goodness-of-fit tests for the message distribution of corresponding ranks in two communication graphs. This identifies both exactly and approximately equivalent computations as seen in Figure 5 and is extremely fast. A significance level allows tuning the false positive and false negative rates of the algorithm. However, swiss-army programs elude this approach.

We plan to investigate classification based on distributions of over-represented subgraphs called *motifs* [25] as well as weighing point-to-point and collective communication separately. Statistical measures such as Claussen's offdiagonal complexity [26] may be useful for approximate topology comparison. Graph edit distance [13], Bayesian [27] and spectral [28] approaches to edit distance, graph kernels [29], and factor graphs [30] offer additional approaches to approximate graph matching.

These directions may provide increased generality for topologies not yet observed or performance bounds in adversarial environments [31]. However, the results presented in this paper show that error-tolerant methods for matching parallel communication patterns are practical for inferring latent classes of computation.

6. ACKNOWLEDGEMENTS

Thanks to Scott Campbell and David Skinner for capturing IPM data at NERSC and to members of the Cyber Security project at LBNL for helpful discussions. This research was supported in part by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy, under contract number DE-AC02-05CH11231, and also by the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001 under the auspices of the Institute for Information Infrastructure Protection (I3P) research program. The I3P is managed by Dartmouth College. The views and conclusions contained in this document are those of the authors and not necessarily those of its sponsors.

7. REFERENCES

[1] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, “State-of-the-art in heterogeneous computing,” *Scientific Programming*, vol. 18, no. 1, pp. 1–33, 2010.

[2] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, “MPI on a Million Processors,” in *Proceedings of the 16th European PVM/MPI Users' Group Meeting*, pp. 20–30, 2009.

[3] R. Metzger and Z. Wen, *Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization.* MIT Press, 2000.

[4] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan, “Transforming MPI source code based on communication patterns,” *Future Generation Computer Systems*, vol. 26, no. 1, pp. 147–154, 2010.

[5] J. Borrill, J. Carter, L. Oliker, D. Skinner, and R. Biswas, “Integrated Performance Monitoring of a Cosmology Application on Leading HEC Platforms,” in *Proceedings of the 2005 International Conference on Parallel Processing*, pp. 119–128, 2005.

[6] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, “The landscape of parallel computing research: A view from Berkeley,” 2006.

[7] P. Colella, “Defining Software Requirements for Scientific Computing,” 2004.

[8] L. Freeman, “A Set of Measures of Centrality Based on Betweenness,” *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977.

[9] D. B. West, *Introduction to Graph Theory.* Prentice Hall, 2nd ed., 2001.

[10] J. R. Ullmann, “An Algorithm for Subgraph Isomorphism,” *Journal of the ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[11] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, “A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[12] A. A. Hagberg, D. A. Schult, and P. J. Swart, “Exploring Network Structure, Dynamics, and Function using NetworkX,” in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), pp. 11–16, 2008.

[13] A. Sanfeliu and K. Fu, “A distance measure between attributed relational graphs for pattern recognition,” *IEEE Transactions on Systems, Man and Cybernetics*, vol. 13, no. 3, pp. 353–362, 1983.

[14] J. D. Gibbons and S. Chakraborti, *Nonparametric Statistical Inference.* CRC Press, 5h ed., 2010.

[15] F. J. Massey, “The Kolmogorov-Smirnov Test for Goodness of Fit,” *Journal of the American Statistical Association*, vol. 46, no. 253, pp. 68 – 78, 1951.

[16] W. J. Conover, “A Kolmogorov Goodness-of-Fit Test for Discontinuous Distributions,” *Journal of the American Statistical Association*, vol. 67, no. 339, pp. 591–596, 1972.

[17] K. Fürlinger, N. J. Wright, and D. Skinner, “Effective Performance Measurement at Petascale Using IPM,” in *Proceedings of the 16th IEEE International Conference on Parallel and Distributed Systems*, pp. 373–380, 2010.

[18] J. Shalf, S. Kamil, L. Oliker, and D. Skinner, “Analyzing Ultra-Scale Application Communication Requirements for a Reconfigurable Hybrid Interconnect,” in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005.

[19] R. Riesen, “Communication Patterns,” in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, pp. 275–282, 2006.

[20] C. Ma, Y. M. Teo, V. March, N. Xiong, I. R. Pop, Y. X. He, and S. See, “An approach for matching communication patterns in parallel applications,” in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–12, 2009.

[21] S. Peisert, “Fingerprinting Communication and Computation on HPC Machines.” 2010.

[22] S. Whalen, *Security Applications of the e-Machine.* PhD thesis, University of California, Davis, 2010.

[23] A. Basumallik and R. Eigenmann, “Towards automatic translation of OpenMP to MPI,” in *Proceedings of the 19th International Conference on Supercomputing*, pp. 189–198, 2005.

[24] A. Basumallik, S. Min, and R. Eigenmann, “Programming Distributed Memory Sytems Using OpenMP,” in *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 207–214, 2007.

[25] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, “Network motifs: Simple building blocks of complex networks,” *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[26] J. C. Claussen, “Offdiagonal complexity: A computationally quick complexity measure for graphs and networks,” *Physica A*, vol. 375, pp. 365–373, Feb. 2007.

[27] R. Myers, R. C. Wison, and E. R. Hancock, “Bayesian graph edit distance,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 6, pp. 628–635, 2000.

[28] A. Robles-Kelly and E. R. Hancock, “Graph edit distance from spectral seriation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 3, pp. 365–378, 2005.

[29] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, “Weisfeiler-Lehman Graph Kernels.” 2010.

[30] J. Reichardt, R. Alamino, and D. Saad, “The interplay of microscopic and mesoscopic structure in complex networks.” 2010.

[31] M. Barreno, P. L. Bartlett, F. J. Chi, A. D. Joseph, B. Nelson, B. I. P. Rubinstein, U. Saini, and J. D. Tygar, “Open problems in the security of learning,” in *Proceedings of the 1st ACM Workshop on AISec*, pp. 19–26, 2008.