PERFORMANCE OF GENERAL GRAPH ISOMORPHISM ALGORITHMS[1]

S. Voss[2], Jaspal Subhlok

Department of Computer Science
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

Technical Report Number UH-CS-09-07

Revised: Mar 24, 2010

Keywords: Graph Isomorphism, Performance Skeletons

Abstract

Graphs are a powerful tool used in pattern matching and graph matching is an isomorphism problem. A significant number of graph isomorphism algorithms are presented in literature, but few papers characterize their performance. Consequently, it is not known how the algorithms will differ as the type, size, and node labeling of the graphs vary in real-world applications. This paper introduces an application for graphs identifying communication topologies for performance skeletons. Then, a benchmarking activity for four exact graph isomorphism algorithms on well-structured graphs is presented. The sizes considered for each topology range from as few as eight nodes up to 16,000 nodes. The affect of node labeling is also explored.

[1] Support for this work was provided by the National Science Foundation under Award No. SCI-0453498 and. CNS-0410797. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[2] S. Voss is with Coe College

PERFORMANCE OF GENERAL GRAPH ISOMORPHISM ALGORITHMS[1]

S. Voss[2], Jaspal Subhlok

Abstract

Graphs are a powerful tool used in pattern matching and graph matching is an isomorphism problem. A significant number of graph isomorphism algorithms are presented in literature, but few papers characterize their performance. Consequently, it is not known how the algorithms will differ as the type, size and node labeling of the graphs vary in real-world applications. This paper introduces an application for graphs identifying communication topologies for performance skeletons. Then, a benchmarking activity for four exact graph isomorphism algorithms on well-structured graphs is presented. The sizes considered for each topology range from as few as eight nodes up to 16,000 nodes. The affect of node labeling is also explored.

Index Terms

Graph Isomorphism, Performance Skeletons

I. INTRODUCTION

Graphs are commonly used to provide structural and/or relational descriptions. A node in a graph can be anything as long as it can be evaluated either quantitatively or qualitatively. The relational information is stored in the set of edges [8]. From the point of view of pattern analysis and recognition, the most important graph-processing challenge is matching graphs for comparison [3]. One of the most relevant sub-problems in this category is matching a sample graph with a reference graph [6].

There is no known polynomial time algorithm for graph isomorphism, and it is well known that subgraph isomorphism is an NP-complete problem. It is still an open question whether graph isomorphism is NP-complete. The exponential time requirements of isomorphism algorithms have been a major obstacle for applications that require the use of large graphs.

There have been considerable research efforts towards improving the performance of isomorphism algorithms in terms of computational time and memory requirements. Some algorithms manage to reduce computational complexity by imposing topological restrictions (e.g. planar graphs [7], and trees [1]). However, in many circumstances graphs are not guaranteed to fall into any specific topological category.

When deciding to work with a graph-based method, choosing an appropriate algorithm for the given application is essential. There are only a few papers [6, 2] comparing the general graph isomorphism algorithms in terms of key performance indices, such as time requirements. This paper examines five general graph isomorphism algorithms and compares their time requirements over a variety of topologies, sizes, and node labelings.

II. MOTIVATION AND CONTEXT

Performance prediction of a parallel application is challenging for foreign environments and is difficult to model. A performance skeleton is a short running program that models the dominant communication structure and computational behavior of a parallel application. Monitored execution of a performance skeleton in a new environment gives an estimate of the performance of the original application this environment. The execution time of the performance skeleton only needs to be scaled to obtain the estimated execution time of the application. The basic procedure for constructing a performance skeleton consists of 1) generating a trace for each process, 2)

[1] Support for this work was provided by the National Science Foundation under Award No. SCI-0453498 and. CNS-0410797. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[2] S. Voss is with Coe College.

converging the traces into a single logical program trace, 3) compressing the logical trace by identifying the loop structure, and 4) converting this information into an executable program [14].

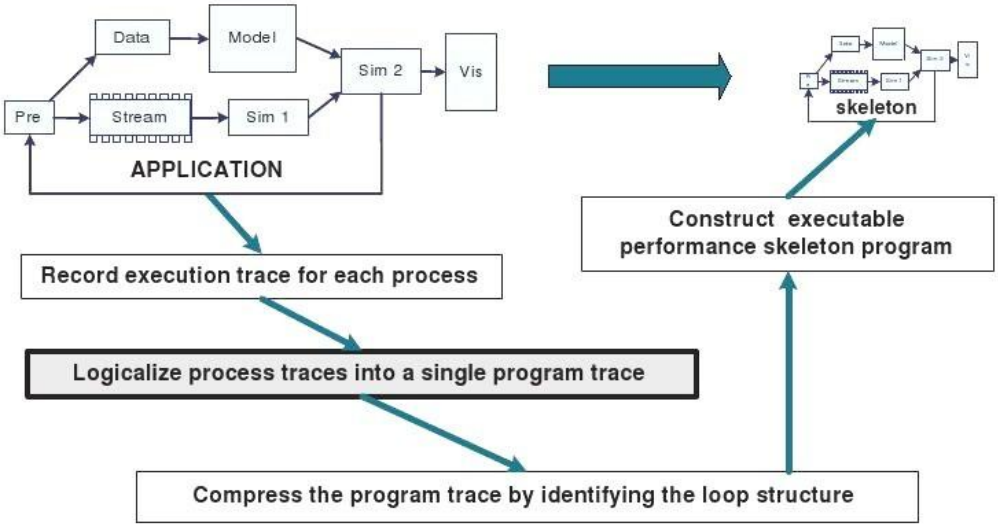

Fig. 1. Performance Skeleton Construction. Map of the construction procedure for performance skeletons.

The highlighted logicalization step in Fig. 1 is the focus of [13]. Logicalization converges the individual process traces into a single logical program trace. To obtain the logical trace the underlying communication structure must be identified. The process traces are represented by an adjacency matrix and compared with a library of known topologies. The topologies are narrowed down based on a set of invariants such as the number of nodes, the number of edges, the degree of the nodes, and the set eigenvalues of the communication matrix. After these invariants are tested, only a few library topologies remain, and in many cases only one topology remains as a candidate for the communication structure. Once the candidate topologies are identified, exact graph matching must be performed to guarantee the match. Exact graph matching is an isomorphism problem.

In the performance skeleton research, the VF2 algorithm was chosen to do the exact graph matching; however, the runtime of the algorithm was slower than expected in some cases. This study was created to explore factors that may affect the execution of the algorithm. One of the main theories to explore was whether process labeling affects runtime.

III. Graph Isomorphism Algorithms

There is no guarantee the communication structure of a parallel application will follow any specific topological category. Therefore the focus of this study is exact graph isomorphism algorithms that do not impose topological restrictions. General graph isomorphism algorithms include: Ullmann's algorithm [12], Schmidt & Druffel's algorithm (SD) [11], the VF algorithms [4, 5], and the Nauty algorithm [10].

The Brute-force solution to graph isomorphism results in a depth-first search tree. Ullmann's algorithm reduces the search space through backtracking. This produces a reduction in the number of successor nodes to be searched. Ullmann's algorithm is designed for both graph isomorphism and subgraph isomorphism. At best Ullmann's algorithm has a time complexity of $O(N^3)$ and its worst case is $O(N!\cdot N^3)$. Spatially the algorithm's complexity is $O(N^3)$.

SD is another backtracking algorithm. It relies on information contained in the distance matrix representation of the graphs. The distance matrix representation is used to establish the initial partition of the graph for the algorithm. It is also used during backtracking to reduce the possible search tree mappings. $O(N^3)$ time is required to generate the distance matrix so the overall time requirement of the algorithm cannot be less than N^3. Given the distance matrix, the initial partition can be realized in $O(N^2)$ time. The backtracking time requirement for SD has $O(N^2)$ as a lower bound and $O(N\cdot N!)$ as an upper bound.

VF is based on a depth-first strategy with a set of rules to prune the search tree. The algorithm relies on a "State Space Representation" where a state represents a partial solution to the matching. A transition between states

corresponds to the addition of a new pair of matched nodes. Best case time complexity is $O(N^2)$ and the worst case is $O(N \cdot N!)$. The spatial complexity is $O(N^2)$, which is less than Ullmann's spatial requirements.

VF2 is based on the same concept as VF but stores the information of the state space search in more efficient data structures. The spatial complexity of VF2 is O(N) with a small constant factor. This permits analysis of larger graphs and provides more proficient use of cache memory for medium sized graphs.

McKay's Nauty is built for automorphisms; however, it can perform transformations that reduce the graphs to a canonical form. If the canonical forms of the two graphs are identical then they are isomorphic.

Ullmann, SD, VF, and VF2 are all included in a graph isomorphism library, VFlib2.0. VFlib2.0 was developed at the University of Naples "Federico II." It was chosen because of the ease of integration with C++ and it implements multiple algorithms. Nauty is available online separately.

IV. The Graphs

The algorithms were compared over various topologies, sizes, and node labelings.

A. Topologies

Communications topologies tend to be highly-structured, so ten structured topologies were selected for this study. They are 2D grids, 2D tori, 6-point and 8-point stencils on both 2D topologies, 3D grids, 3D tori, binary trees, and hypercubes. The 2D topologies are square, the 3D topologies cubes, and the binary trees are complete.

1) Stencils, Grids, and Tori

Stencil is a way of referring to the pattern in which a graph's nodes are connected. A 4-point stencil means each node is connected to four nodes. It is important to note that the term 4-point stencil does not refer to a unique graph. The below figure shows two different examples of 4-point stencils.

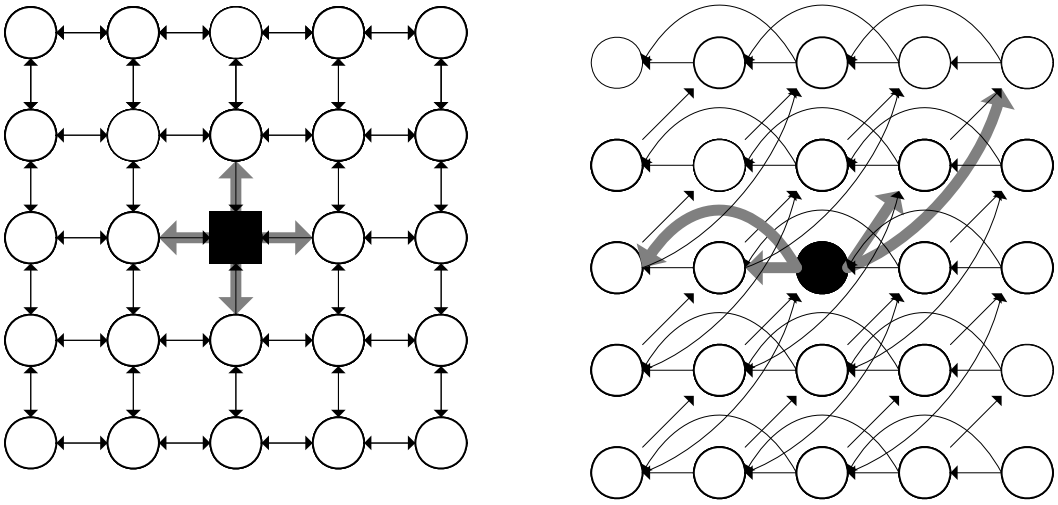

Fig. 2. 4-point stencils. The two figures illustrates the variation in stencils.

In Fig. 2 the grey arrows point to the nodes in which the black node is connected. Each node of the graph is connected in the same way as the black node. The two graphs are both 4-point stencils but are different graphs. The graph on the left is a 2D grid with each node connected to its North, South, West, and East. The graph on the right is connected to the West, two West, Northeast, and two Northeast. The 6-point and 8-point stencils used for this study are shown below.

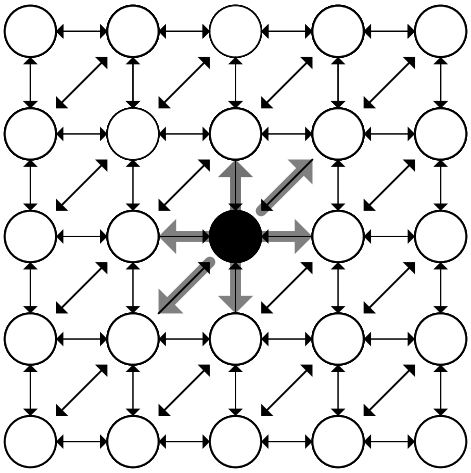

Fig. 3. 6-point stencil.

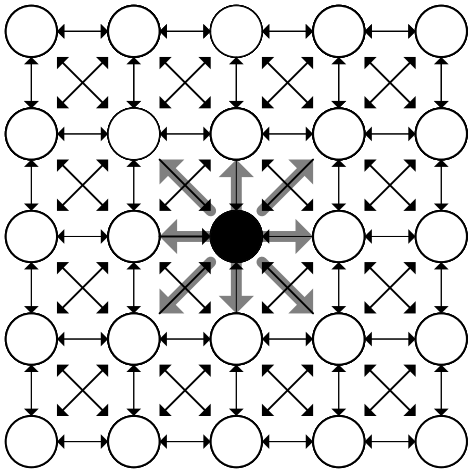

Fig. 4. 8-point stencil.

The 6-point stencil adds the Northeast and Southwest neighbors to underlying pattern while the 8-point stencil also adds the Northwest and Southeast neighbors. The 3D topologies have neighbors North, South, West, East, Up, and Down.

In the context of this paper, maximally connected means a node is connected to all possible neighbors within the pattern. A torus is maximally connected while a grid is not. A node in a 2D torus has North, South, West, and East neighbors. However, in the 2D grid only the nodes in the interior are maximally connected. The nodes on the perimeter (i.e. first and last rows and columns) do not have all neighbors in the pattern. Below, in Fig. 4, examples of a 2D grid and a 2D torus are presented.

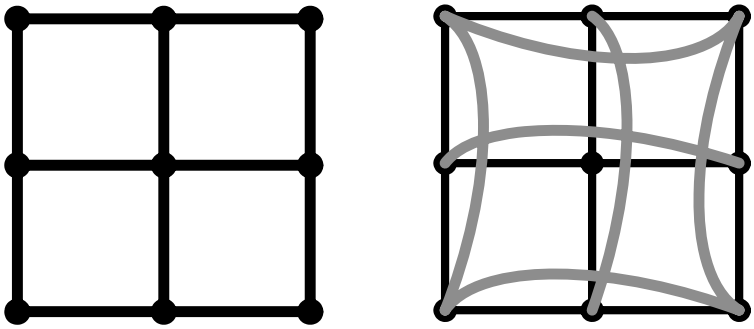

Fig. 5. Grid versus Torus. Illustrates the difference between grids and tori.

The figure on the left is a 9-node 2D grid and the figure on the right is a 9-node 2D torus. The grey lines in the figure on the right show the node connections wrapping around. The grid's connections do not wrap around. Every node in the tori has four connections while only the interior node in the grid has all four connections.

2) Binary Tree

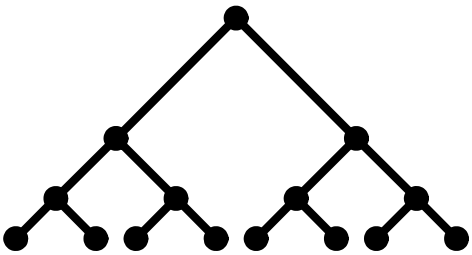

Fig. 6. Complete binary tree.

The binary tree tested is complete, so every node has either zero or two children and all the leaves are at the same depth. A node's neighbors are the parent, the left child, and the right child.

3) Hypercubes

A hypercube is the n-dimensional equivalent of a square (n = 2) and a cube (n = 3). Fig. 7 shows the first five hypercubes. Hypercubes with an 'n' of four or greater were used for this research.

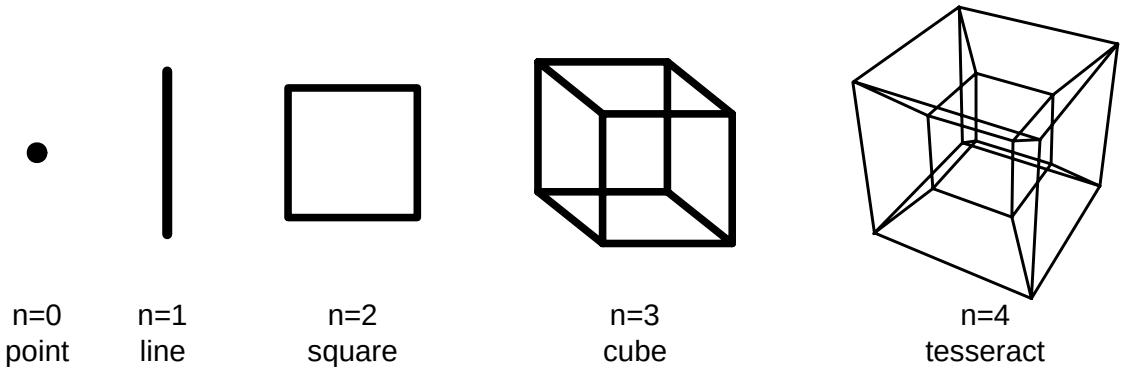

Fig. 7. Hypercubes. First five hypercubes.

B. Sizes

2D topologies range in size from 9-16,641 nodes. The smallest 2D topology tested is 3x3. 3D grids range in size from 27-17,576 nodes. The smallest is 3x3x3. The 27-node (a.k.a. the 3x3x3) 3D tori could not be tested because of the internal way the algorithm holds data. To the data structure it is as if edges are being repeated and this is not handled. The binary trees range in size from 7-16,383 nodes, while the hypercube ranges in size from 8-16,384 nodes. The sizes were incremented such that the next test size is approximately double the previous test case.

C. Node labeling

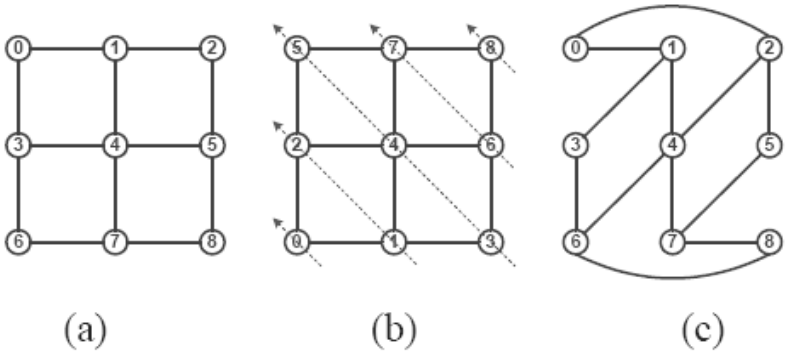

Fig. 8. Node labelings. Illustration of the effect of node labeling.

A topology is easy to identify when the nodes are assigned labels in a well-defined manner, but in general, it is a more difficult problem. This is illustrated by the examples in Fig. 8. The figure shows 9-node 2D grids. Fig. 8(a) shows the underlying 2D grid assigned labels in row-major order. However, in Fig. 8(b) the nodes are numbered diagonally with respect to the underlying 2D grid pattern; starting with the lower left-hand corner. If the graph in Fig. 8(b) were laid out in row major order, it would appear as Fig. 8(c). The underlying 2D grid topology is easily identified in the scenarios represented by Fig. 8(a) and 8(b) but harder to identify in Fig. 8(c). Identification would be even more difficult if the node labeling followed an unknown or arbitrary order.

To present this issue, testing is executed between a well-ordered graph and a graph that has randomness introduced into its labeling. The well-ordered labeling used for the grids and tori is row-major order. The well-ordered labeling used for the binary tree is level-order. For the hypercube, each added dimension to the hypercube duplicates the labeling of the previous dimension and then adds the previous number of nodes to each new node's label in order to keep the labeling unique.

For each size of the topologies, five graphs are created, each with a different degree of randomness: 0, 25, 50, 75, and 99. The degree of randomness is the percent of node labels rearranged within well-ordered labeling. For example, to produce a graph with 25 as its degree of randomness, 25 percent of the node labels within the well-ordered graph are swapped. Degree 0 is the well-ordered version of the graph.

To rearrange the node labels, a non-repeating list of node labels is generated using a provided random function such as Python's *rand*(). The length of the list is determined by the degree of randomness. Pairs of nodes are then selected from the list and the node labels are swapped.

V. Method

Python was used to write the graph creation code. The code creates a well-ordered version of the size of the topology it is making and randomized version. The randomized version are created by swapping a given percent of the well-ordered graph's node labels. The percent of nodes to be swapped can be changed so there is control over the degree of randomness. The python code produces text files containing an adjacency list

The C++ code utilizing VFlib2.0 reads the text files and constructs the graphs using the data structures provided by VFlib. Each of the four isomorphism algorithms provided in VFlib checks the graphs for isomorphism.

For each algorithm to run, a state must be constructed for the algorithm using the two graphs being tested. Timings are taken using gettimeofday() for the initial load of the graphs in VFlib2.0, the construction of the algorithm's state, and the execution of the algorithm itself. The timings are written out to a file to be analyzed.

VI. Nauty

Nauty could not be used to test for isomorphism in this study because of the way isomorphisms were created for this research. Nauty produces the canonical form of graphs. If the canonical form of two graphs are the same then the graphs are isomorphic. This research is interested in whether or not node labeling affects the runtime of the algorithms, so node labels are swapped. When Nauty constructs the canonical form of the graphs, the node labels stay swapped causing Nauty to find the graphs non-isomorphic.

VII. Experimental Results

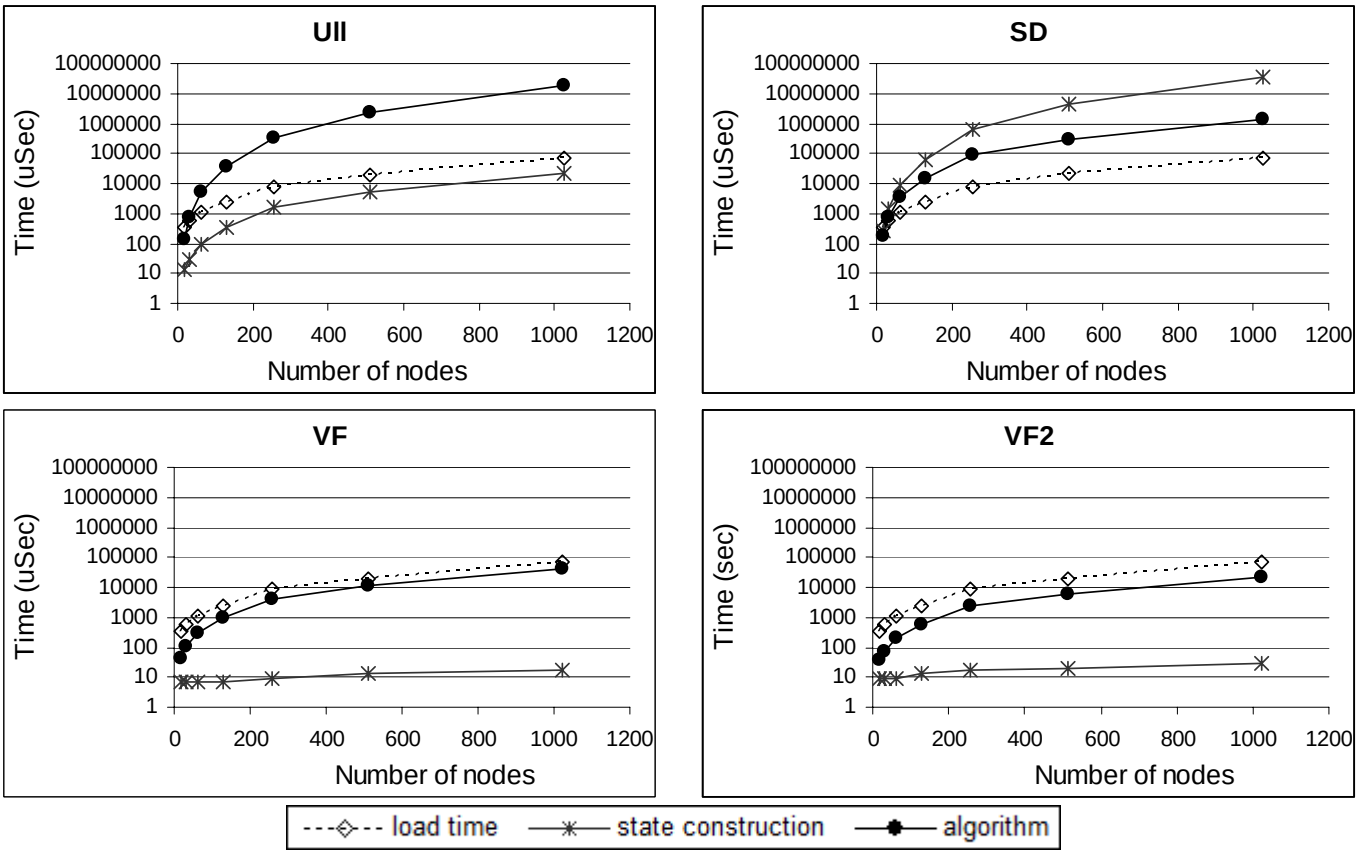

Fig. 9 Complete results on 2D grid.

To use VFlib2.0 the graphs are loaded and stored using built-in data structures. In order to compare two graphs, a state must be constructed from the two graphs for the algorithm to use. Fig. 9 show results of each of the algorithms run on hypercubes with the degree of randomess being 99. The load time for graphs is consistent between all the algorithms. However, the state construction and algorithm times vary.

SD's state construction takes significantly longer than the other algorithms. This could be due to the fact SD relies on information contained in the distance matrix representation of the graphs. This would have to be computed before running the algorithm. Since the state construction varies among the algorithms, the timings presented for the rest of the results in this paper is the combined time of state construction and algorithm runtime.

After 1000 nodes there is insufficient memory for Ullmann to execute. At 1000 nodes, SD takes approximately 35 seconds. Increasing the nodes to 2000, SD takes around 404 seconds (6:44 minutes). This is a 1053% increase. Increasing the nodes to 4000, SD takes around 55 minutes. Due to time constraints six minutes is used as a cut-off for the algorithms. Once a test takes over six minutes, no other tests are run for that algorithm on the given topology. VF and VF2 are able to perform for graphs over 16,000 nodes and larger results for these algorithms will presented in the rest of the paper.

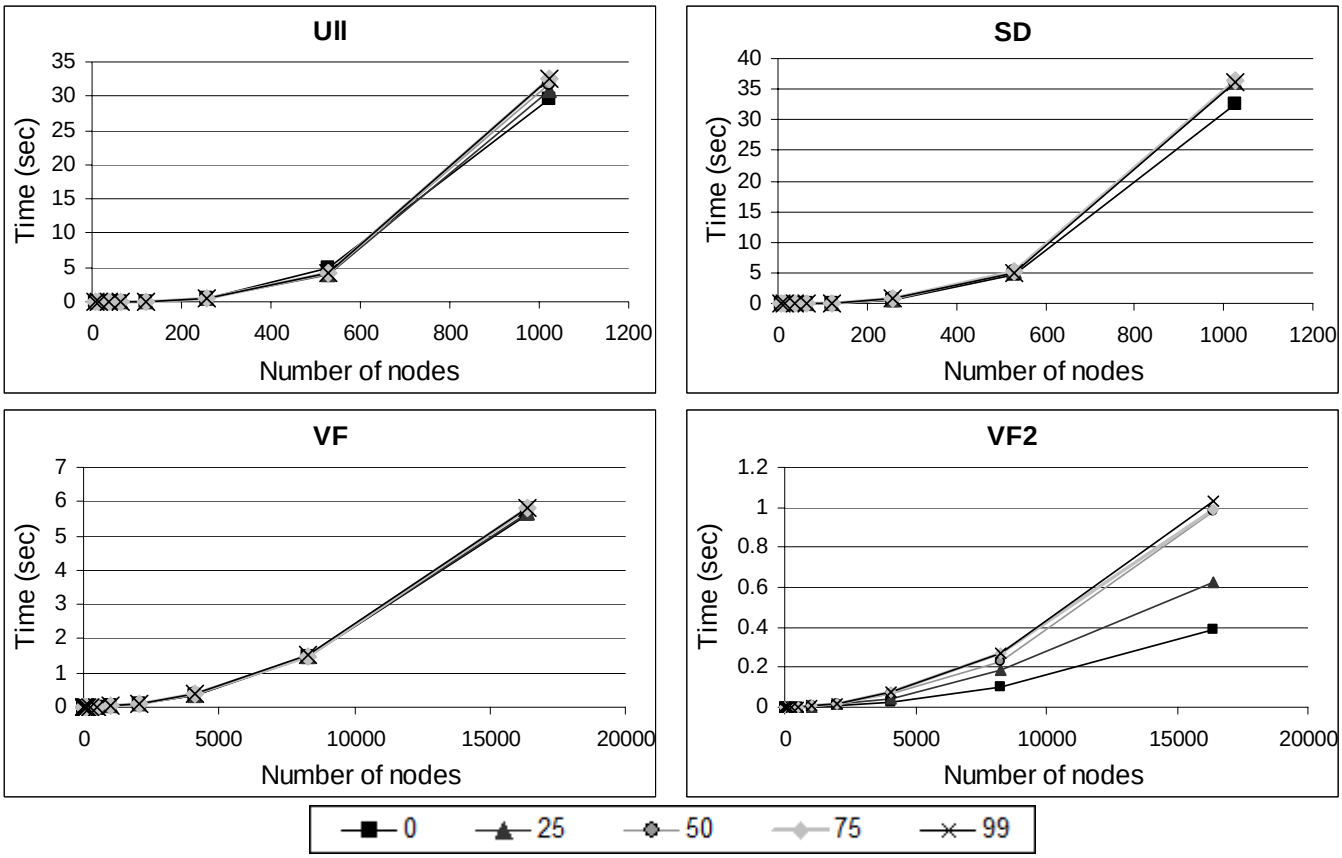

Fig. 10. Results of 2D grid.

Fig. 10 shows the result of the algorithms preformed on 2D grids with varying degrees of randomness. VF and VF2 perform significantly more efficient than Ullmann and SD. Node labeling has little effect on Ullmann, SD, and VF for the 2D grid pattern. Node labeling does have a significant impact on VF2. Without rearranging the node labelsVF2 can execute in 0.358 seconds for 16,384 nodes. With 99% of the node labels swapped, the algorithm takes 1.04 seconds. This is a 190.5% increase just for swapping labels on the nodes.

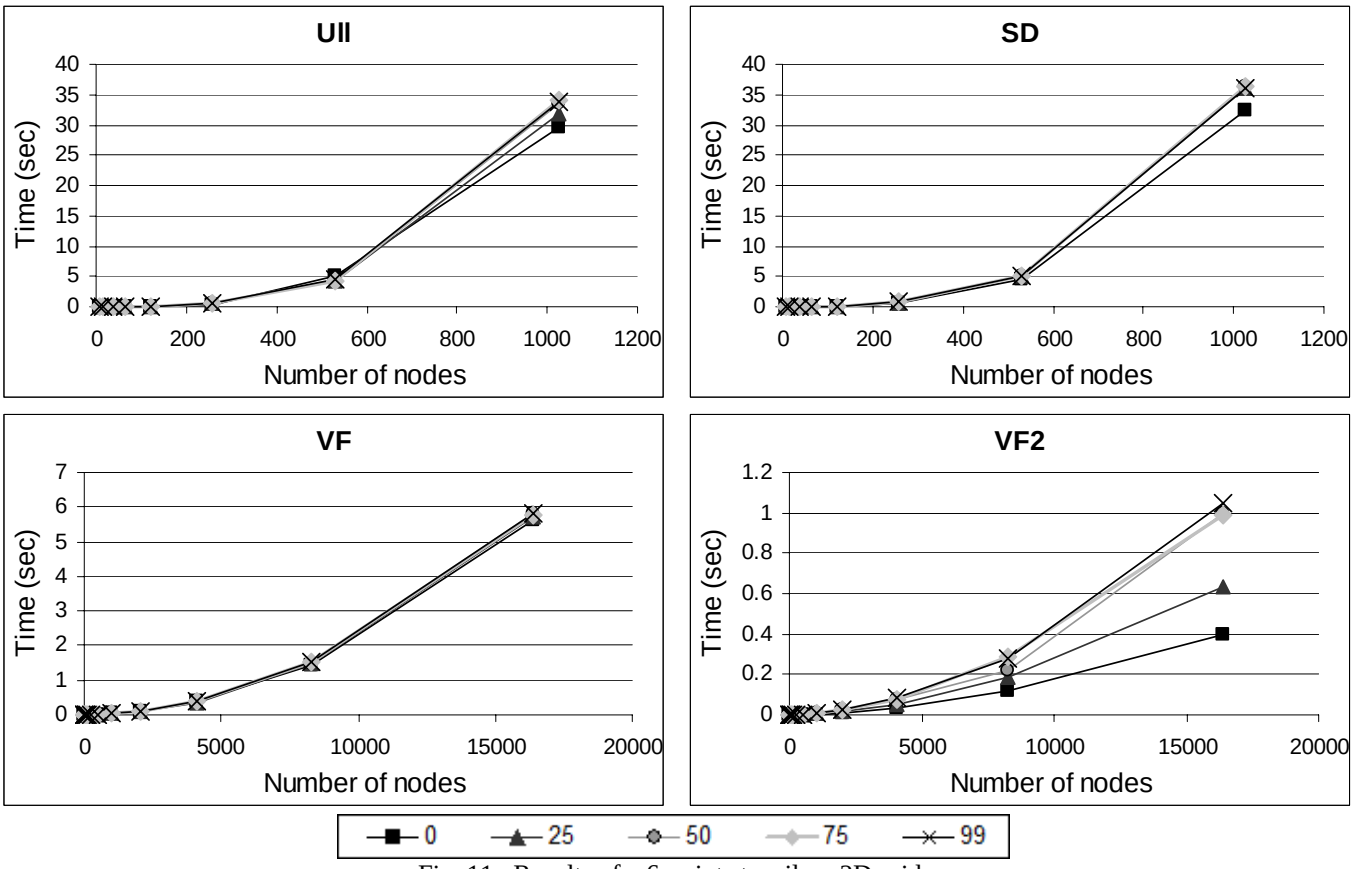

Fig. 11. Results of a 6-point stencil on 2D grid.

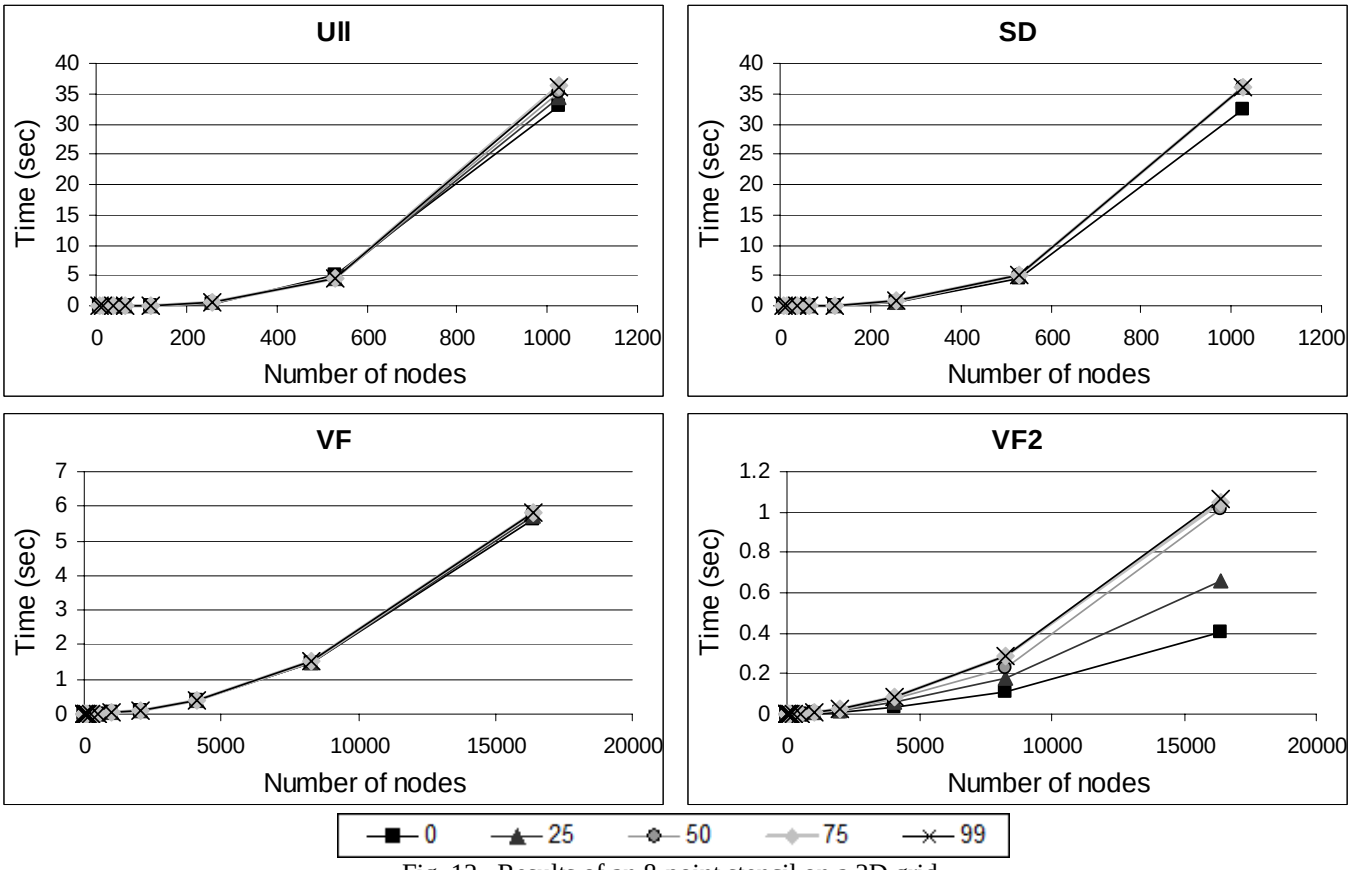

Fig. 12. Results of an 8-point stencil on a 2D grid.

Fig. 11 shows the timings of the algorithms for a 6-point stencil on a 2D grid and Fig. 12 shows results for an 8-point stencil on a 2D grid. The results in Fig. 11 and Fig12 are similar to the results on a 2D grid.

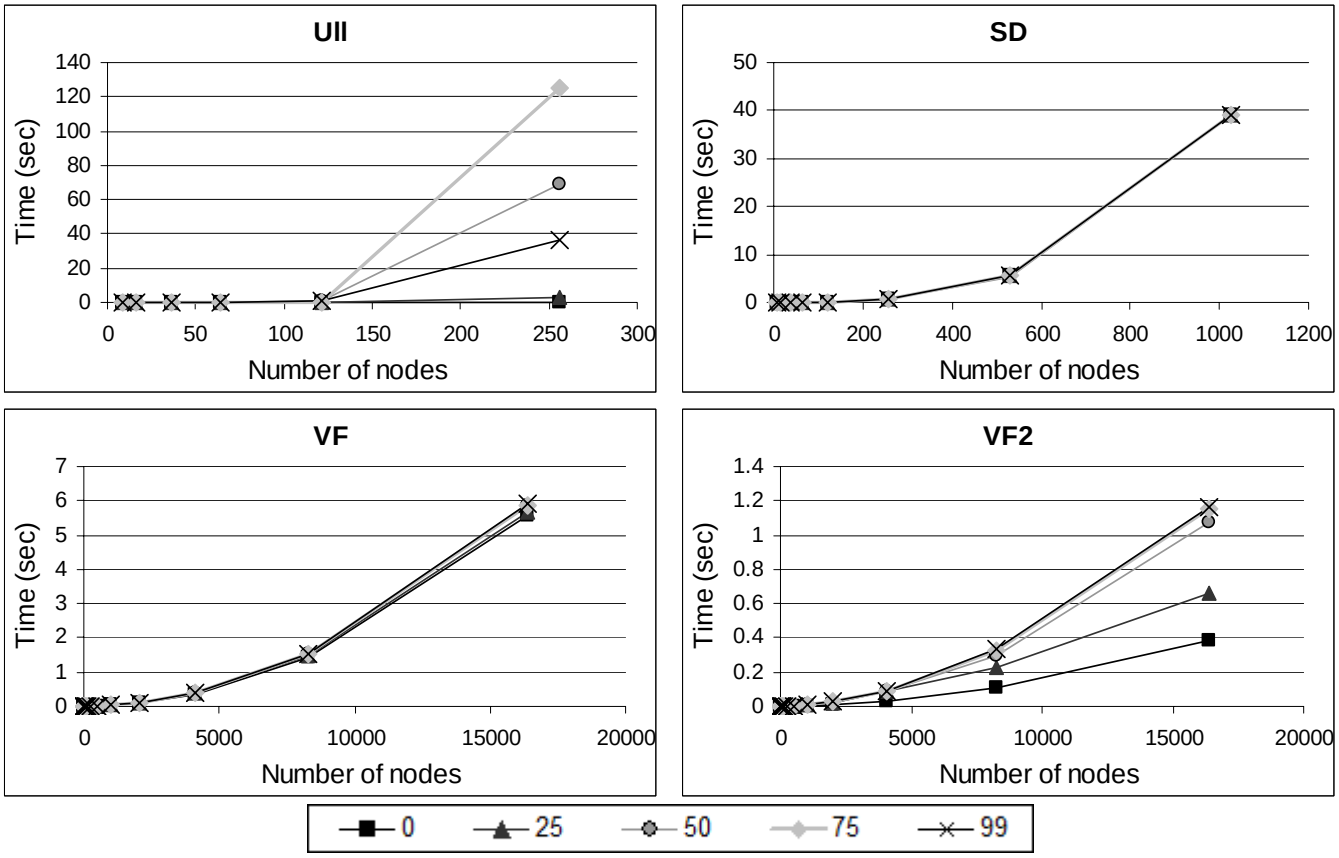

Fig. 13. Results of 2D torus.

Fig. 13 shows SD and VF are both unaffected by node labeling on the 2D torus. VF2 is again affected by node labeling. SD, VF, and VF2 all had timings just slightly higher than their performance on the 2D grid pattern. Ullmann however is strongly affected by node labeling and the topology. In the case of a 2D torus topology, Ullmann is inefficient after 250 nodes so testing of this algorithm had to be terminated. However without node label swapping Ullmann was able to perform though 1,000 nodes.

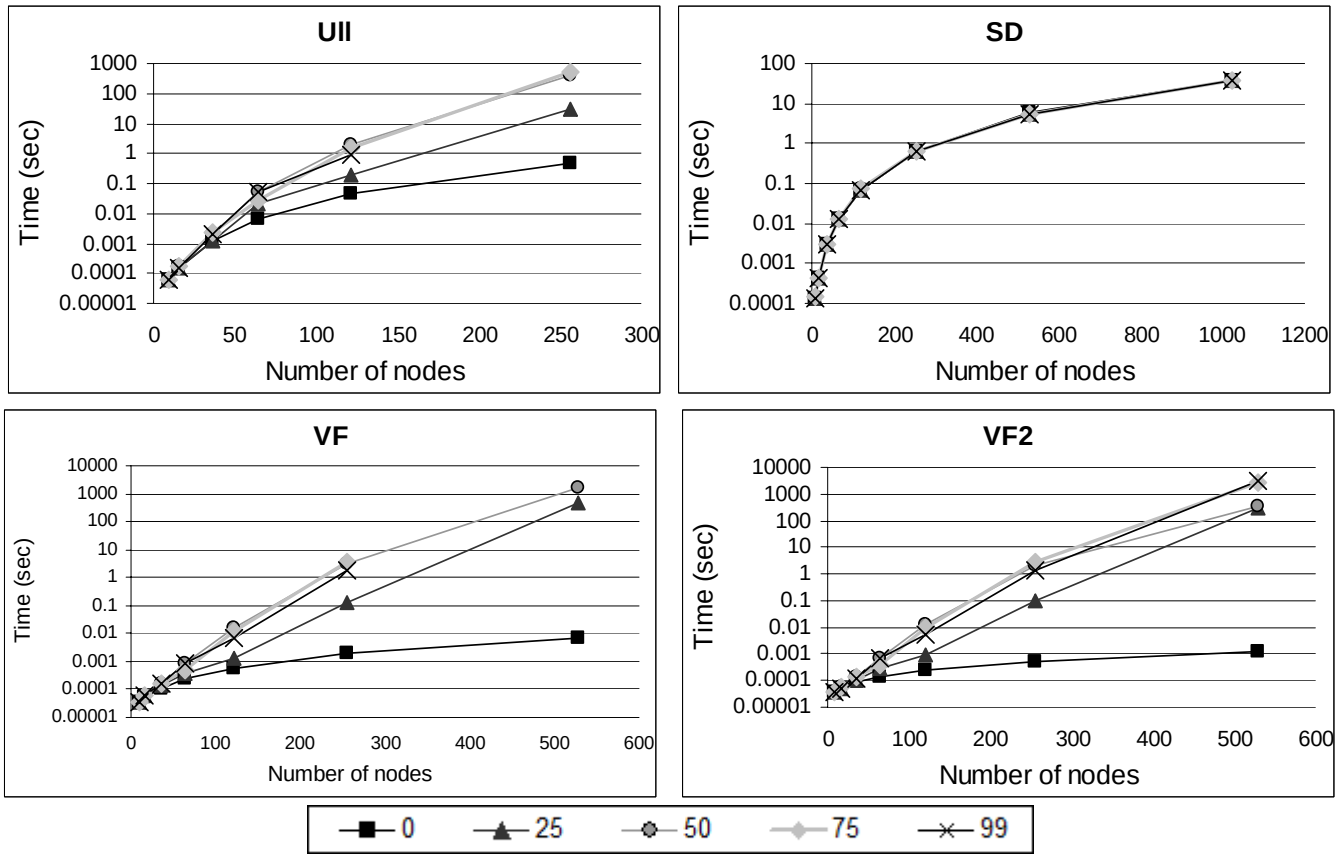

Fig. 14. Results of a 6-point stencil on 2D torus.

Note the graphs in Fig. 14 all have a logarithmic scale. Unlike a stencil being applied to the grid pattern, the 6-point stencil applied to the 2D torus significantly affects all the algorithms except SD. SD was able to finish testing through 1,000 nodes, as it could on the previously presented topologies.

Node labeling again affected Ullmann in this topology. Ullmann successfully tested through 120 nodes. However it struggled through the 250 node test cases and was not able to perform efficiently for the 99 test case. With no node label swapping Ullmann is able to perform though 1,000 nodes.

VF which had been previously been unaffected by node labeling was affected on the 6-point stencil applied to a 2D torus. VF was able to finish complete testing through 250 nodes but could not make though the 50, 75 and 99 test cases within the time limits. VF2 completed testing through 500 nodes but was not always able to finish the 50, 75, and 99 test cases within the time limits. At the 1,000 node test case with 25 percent of the node labels swapped both algorithms were unable to complete in 24 hours. With no node label swapping both algorithms were able to execute though 16,000 nodes within the time limits.

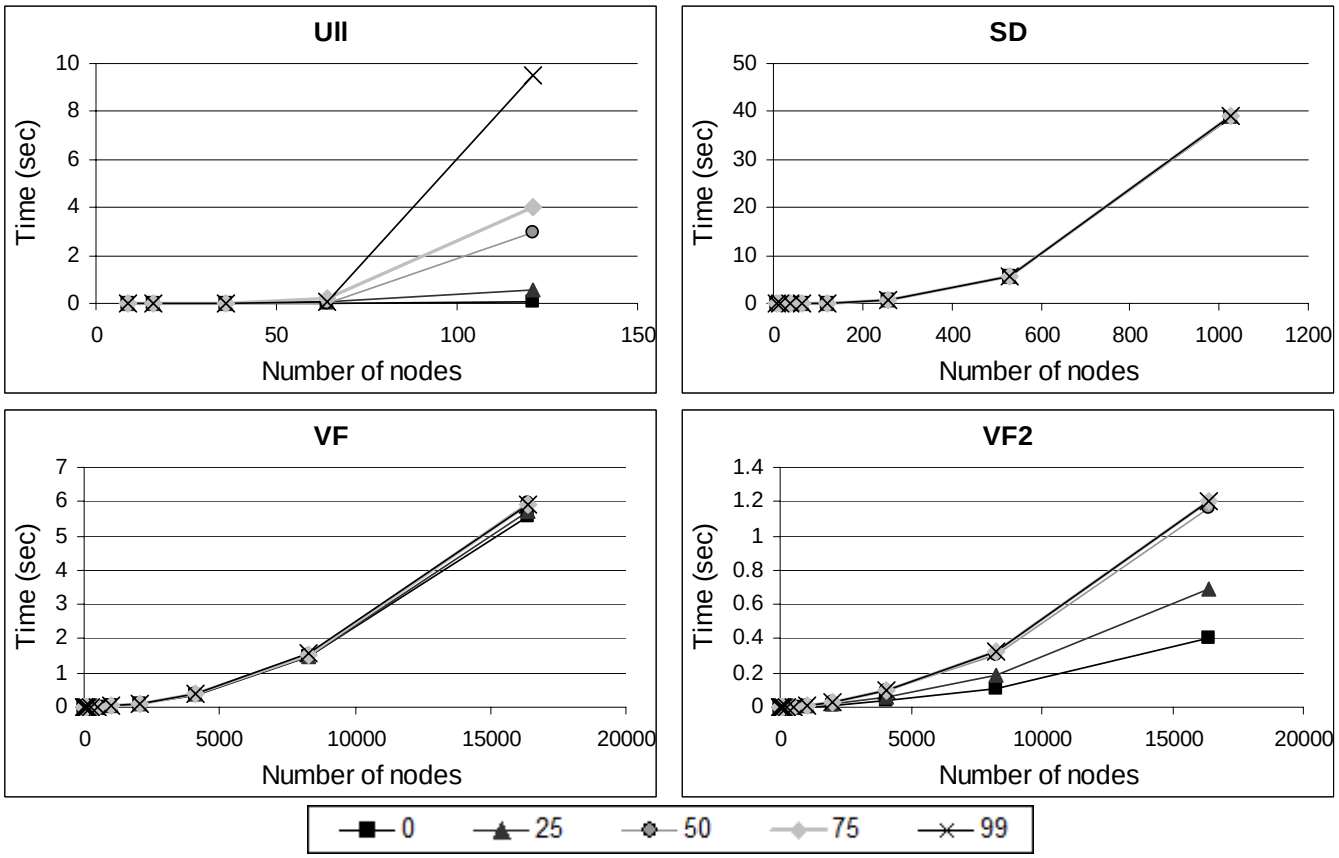

Fig. 15. Results of an 8-point stencil on a 2D torus.

Fig. 15 shows the results of the algorithms applied to an 8-point stencil on a 2D torus. VF, VF2, and SD applied to an 8-point stencil on a 2D torus perform similarly to the way they performed on a 2D torus pattern. Ullmann, however, is significantly affected by this topology and is affected by node labeling. For this topology, Ullmann can only complete testing through 120 nodes efficiently when node label swapping is involved. With no node label swapping Ullmann is able to complete tests though 1,000 nodes.

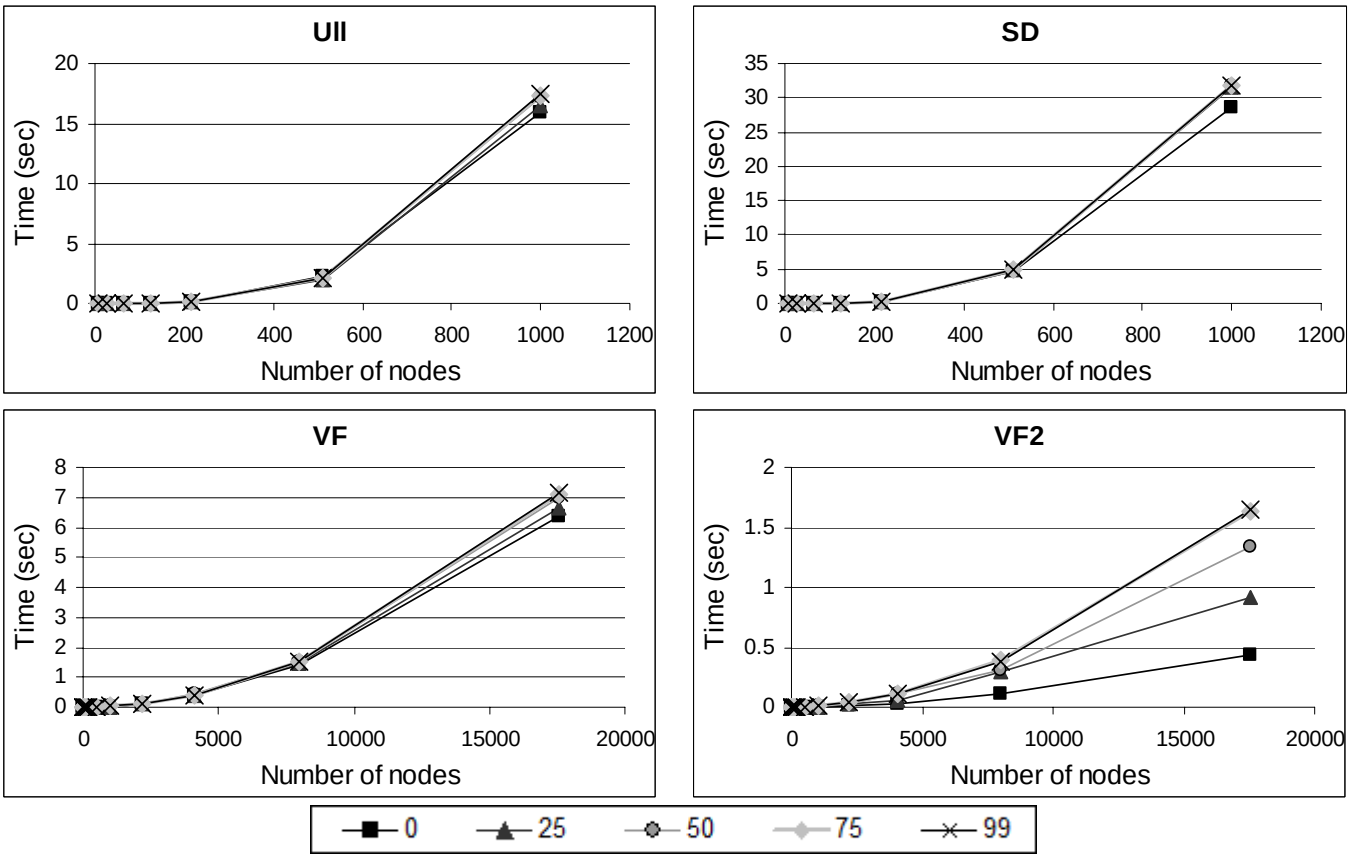

Fig. 16. Results of 3D grid.

Fig. 16 shows the algorithms performed on a 3D grid pattern behaving similarly to the way they performed on a 2D grid pattern. Again, only VF2 is significantly affected by node labeling on a 3D grid pattern.

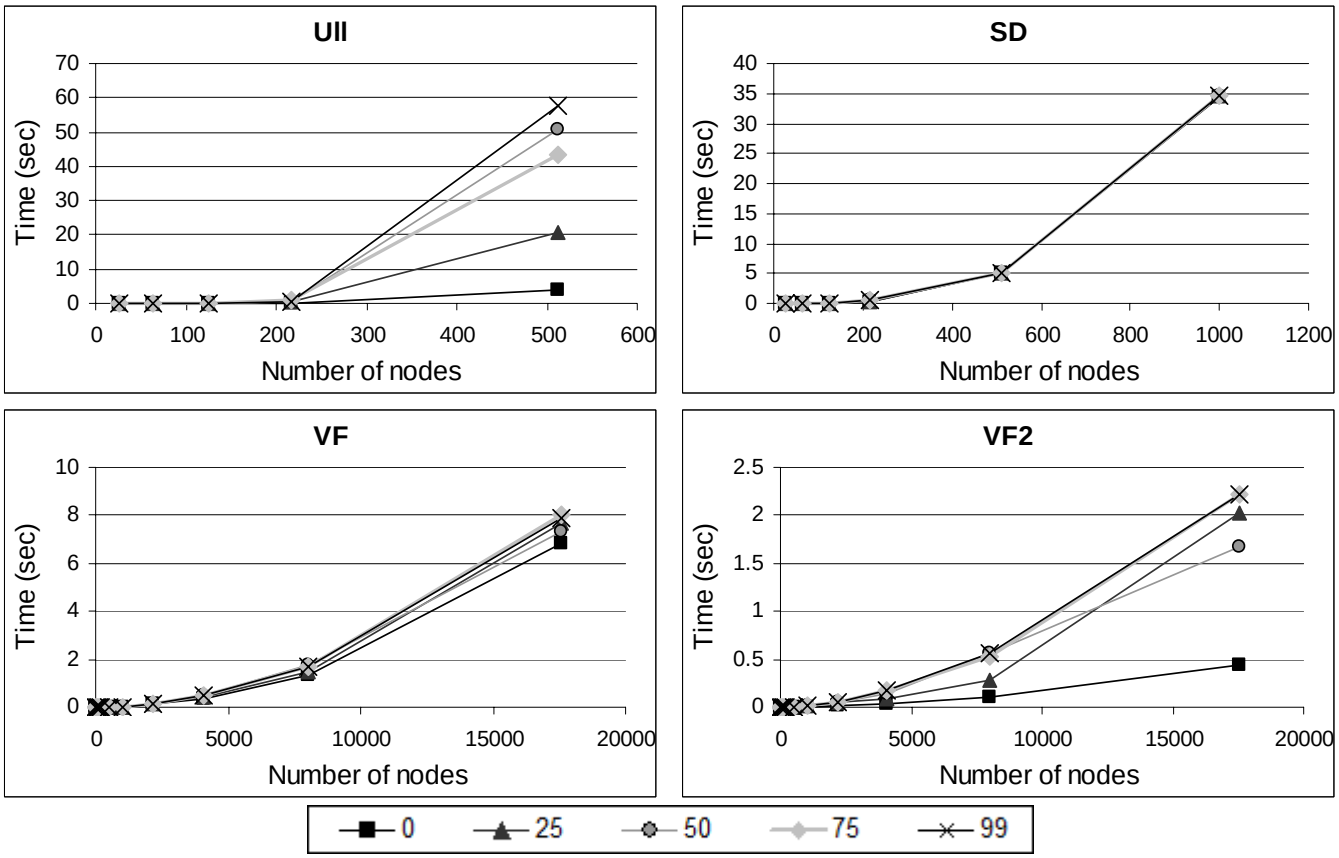

Fig. 17. Results of 3D torus.

Fig. 17 shows that not only VF2 is affected by node labeling, but also Ullmann is also strongly affected by node labeling on a 3D torus pattern. VF is slightly affected by node labeling but not as much as the other two algorithms. Ullmann is only able to complete testing through 500 nodes. Again though, without node label swapping Ullmann could perform though 1,000 nodes.

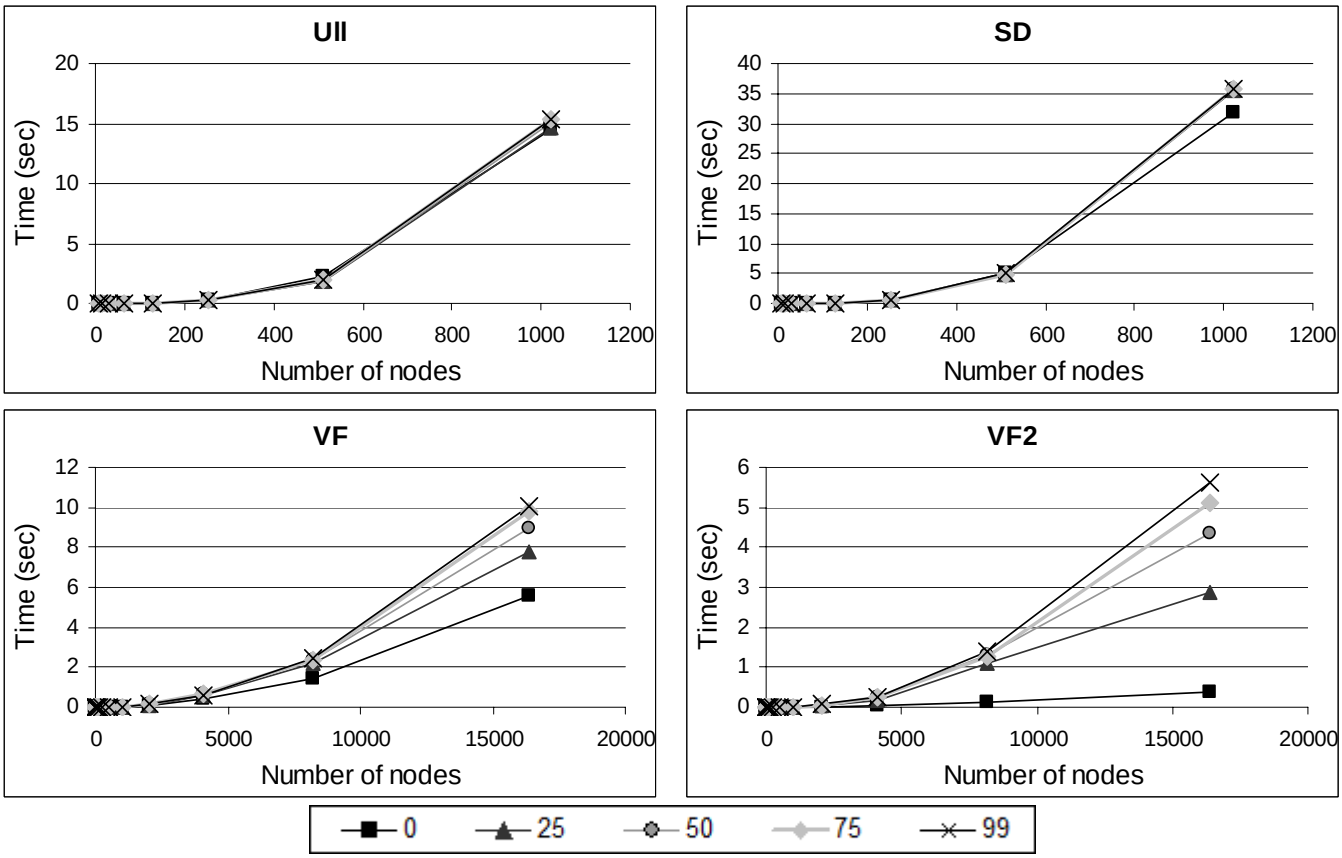

Fig. 18. Results of binary tree.

In Fig. 18 Ullmann and SD are not significantly affected by node labeling on a binary tree pattern while VF and VF2 do appear to be affected. VF and VF2 also run slower on the binary tree pattern than they did on most of the grid patterns.

VIII. Comparison of Algorithms

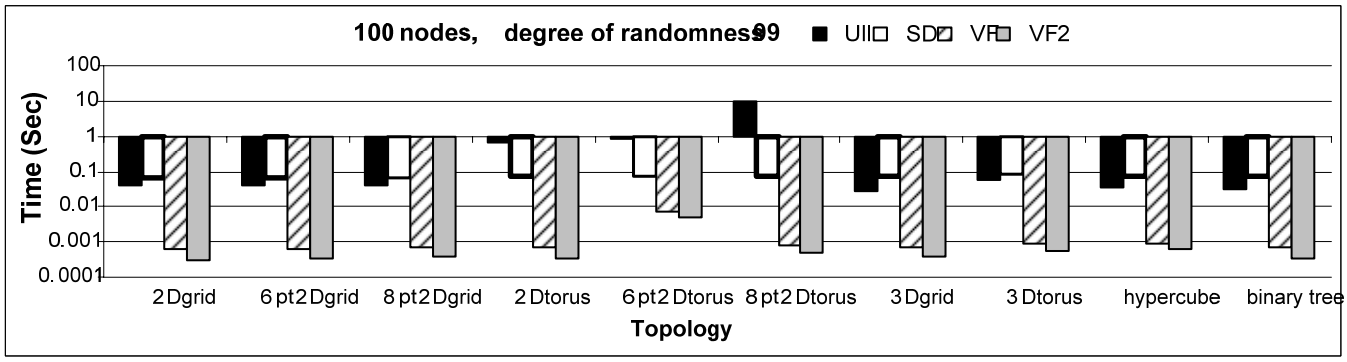

Fig. 19. Performance of algorithms by topology, size 100.

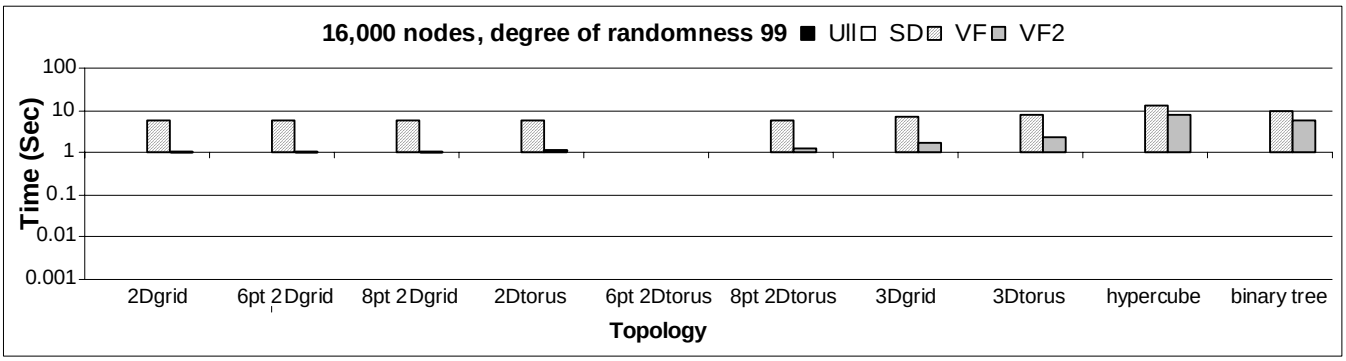

Fig. 20. Performance of algorithms by topology, size 1000. Missing data points refer to instances where execution could not be completed on the reference machine employed for at least 6 hours.

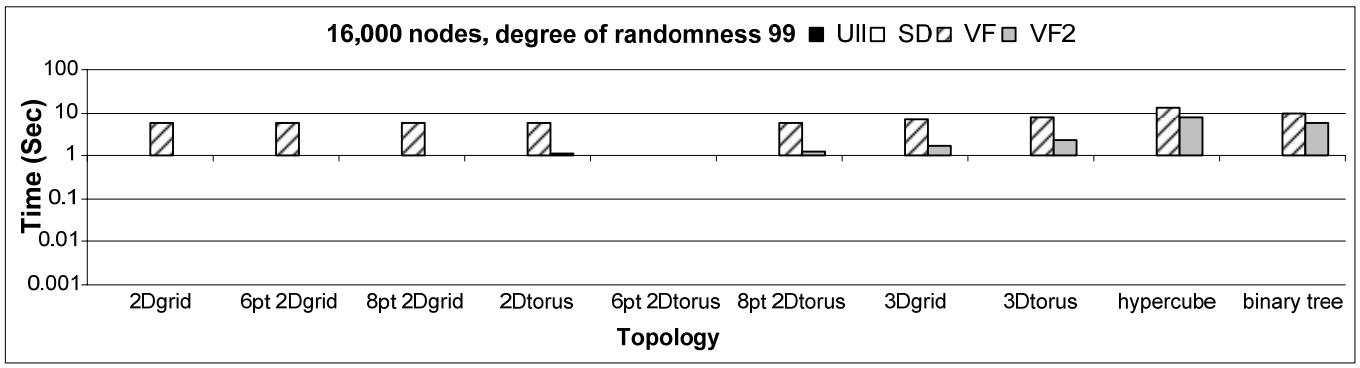

Fig. 21. Performance of algorithms by topology, size 16,000. Missing data points refer to instances where execution could not be completed on the reference machine employed for at least 30 hours.

In each of the three preceding graphs, the degree of randomness is fixed at 99, so the algorithms affected by node labeling are penalized. Note the time scale is logarithmic in microseconds.

One hundred nodes was the largest test size all the algorithms were able to complete for each topology. In Fig. 19 it can be seen that all algorithms were able to execute in less than a second for 100 nodes. VF2 was fastest followed by VF, then Ullmann and SD. In all cases except the 6-point stencil on a 2D torus, VF and VF2 ran in under a millisecond. SD was the most consistent across all the topologies while Ullmann was strongly affected by topology, as a torus pattern with node label swapping took significantly longer than a grid pattern.

Fig. 20 shows the results for 1,000 nodes. VF and VF2 are again fastest and are able to execute in less than a second for all cases except the 6-point stencil applied to the 2D torus. In this case, SD is fastest after 500 nodes. Ullmann is not able to perform efficiently on torus topologies at the 1,000-node level. At 16,000 nodes, only VF

and VF2 are still able to perform and they run efficiently on all topologies tested except the 6-point stencil on the 2D torus.

IX. Discussion and Conclusions

VF2 is the most efficient algorithm followed by VF. However, both are affected by topology. For some topologies, Ullmann and the VF algorithms would not complete on a typical workstation given a long duration of time. VF2 was the only algorithm affected by node labeling for all topologies, while VF and Ullmann were only affected by node labeling on a few topologies. Ullmann is faster than SD in cases where Ullmann is not affected by topology. SD is the most consistent as it is not affected by either topology or node labeling in any test case.

The 6-point stencil was the only not reflectively symmetric topology tested. After seeing the results of VF and VF2 applied to the 6-point stencil on the 2D torus other non-reflective symmetric stencils were built on the 3D grid and 3D tori. Some slow down of the algorithms was seen on these topologies, but not nearly to the extreme of the 6-point stencil. More research needs to be done on how these algorithms work and why they are affected by topologies and node labeling. However the best algorithm for a given situation can be found by combing the results in [6, 2], which test mostly unstructured graphs, with this paper which focuses on highly structured graphs.

X. References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullmann, “The design and analysis of computer algorithms,” Addison Wesley, 1974.

[2] H. Bunke, M.Vento, “Benchmarking of Graph Matching Algorithms,” Proceedings 2nd IAPR-TC15 Workshop on Graph-based Representations, Handhorf, 1999.

[3] L.P. Cordella, P. Foggia, C. Sansone, 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.

[4] L.P. Cordella, P. Foggia, C. Sansone, M. Vento, “Performance Evaluation of the VF Graph Matching Algorithm,” Proceedings of the 10th ICIAP, IEEE Computer Society Press, pp. 1172-1177, 1999.

[5] P. Foggia, C. Sansone, M. Vento, “An Improved Algorithm for Matching Large Graphs,” Proceedings of the 3rd IAPR-TC15 International Workshop on Graph-based Representation, Italy, 2001.

[6] P. Foggia, C. Sansone, M. Vento, “A Performance Comparison of Five Algorithms for Graph Isomorphism,” Proceedings of the 3rd IAPR-TC15 Workshop on Graph based Representation (gbr2001), Italy, 2001.

[7] J. Hopcroft, J. Wong, “Linear time algorithm for isomorphism of planar graphs,” Proceedings of the 6th annual ACM Sym. Of Theory of Computing, pp. 162-184, 1974.

[8] M.J. Jolion, “Graph matching: what are we really talking about?,” Third IAPR Workshop on Graph-based Representations in Pattern Recognition, Ischia, Italy, May 2001. Http://rfv.insa-lyon.fr/~jolion/PS/prlcp1.pdf

[9] E.M. Luks, “Isomorphism of Graphs of bounded valence can be tested in polynomial time,” Journal of Computer System Science, pp. 42-65, 1974.

[10] B.D. mckay, “Practical Graph Isomorphism, Congressus Numerantium, 30, pp.45-87,1981.

[11] D.C. Schmidt, L.E. Druffel, “A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices,” Journal of the Association for Computing Machinery, 23, pp. 433-445, 1976.

[12] J.R. Ullmann, “An Algorithm for Subgraph Isomorphism,” Journal of the Association for Computing Machinery, vol. 23, pp. 31-42, 1976.

[13] Q. Xu, R. Prithivathi, J. Subhlok, R. Zheng, “Logicalization of {MPI} Communication Traces,” Technical Report UH-CS-08-07, University of Houston, May 2008.

[14] Q. Xu and J. Subhlok, Construction and Evaluation of Coordinated Performance Skeletons, The 15th annual IEEE International Conference on High Performance Computing (HiPC 2008), Bangalore, India, December 2008. Published as Springer-Verlag LNCS 5374, pp 73-86.