SAND2015-1944C

Performance Analysis of a Program Understanding Static Analysis Signature Search Algorithm

Aditya M. Deshpande and Jeffrey T. Draper
Information Sciences Institute
University of Southern California,
Los Angeles, CA USA
amdeshpa@usc.edu, draper@isi.edu

J. Brian Rigdon and Richard F. Barrett
Sandia National Laboratories
Albuquerque, NM, USA
rfbarre, jbrigdo@sandia.gov

***Abstract*—Graph processing is widely used in data analytics applications in a variety of fields, and is rapidly gaining attention in the computational scientific and engineering (CSE) application community. These applications typically exhibit runtime profiles distinct from those of CSE applications. PathFinder, a proxy for real-world graph analytics applications, searches directed graphs to find characteristic signatures, which are specified sequences of labels that are associated with nodes in the graph.**

In this work, we describe PathFinder and its general performance traits, including its cache behavior, scaling properties, and the impact of increase in graph size density. We highlight some important differences between PathFinder and traditional CSE applications.

***Index Terms*—data analytics, computational science and engineering applications; high performance computing; programming models.**

I. Introduction

In the computer science field of program understanding, static analysis is used to determine the control and data flow of a compiled binary without executing that binary. Data flow seeks to identify the manner in which data is operated on. Our focus here is on control flow, which seeks to identify the possible paths through the program. This can be defined as a directed graph, where edges show the flow between nodes.

Graphs generated from binaries have several interesting properties not seen in other sorts of graphs. An application proxy, named PathFinder, represents these properties, allowing examination of the performance characteristics of the algorithms used in this work. In particular, our goal is to find signatures in the control flow, for example: the first, the shortest, or some statistical measure of the total number of signatures.

In this paper we examine the performance aspects of PathFinder, a proxy for full applications. We compare these with those of some traditional computational science and engineering (CSE) applications.

A. Related Work

Graphs provide a powerful means for describing relationships within physical systems across a broad set of domains. CSE application programs organize unstructured grid points as undirected graphs for spatial decomposition across parallel processes [5], [7]. Social networks use graphs to determine relationships between members, where again, the graphs are undirected since the relationship is presumed to go both ways. Betweenness centrality measures the number of shortest paths through a node [6], where a high value conveys something about the importance of a node. The graph may be directed or undirected. PathFinder is designed to investigate control flow using a directed graph. So in addition to node connectivity, the actual execution path is important. That is, questions pertain not simply to if nodes are connected, but how they are connected.

The Graph500 benchmark (www.graph500.org) operates on graphs such as those defined by social networks. Using a breadth-first search of large undirected graphs, an ordering of computer performance is based on the number of edges traversed (TEPS). The Scalable Synthetic Compact Applications (SSCA) benchmark #2 [2] represents betweenness centrality, where the graph may be directed or undirected. Although strong support for work in each of these areas would generally have a positive impact on the performance of PathFinder, PathFinder algorithms are sufficiently different to warrant a separate effort.

The characteristics of the graphs being operated on are different in ways that impact the runtime characteristics of the algorithms operating on them. Graphs in these other areas typically have a small diameter with a large degree, thus leading to the notion of "six degrees of separation." PathFinder graphs are typically long and skinny, "look very directional", and thus have large diameters with small degrees.

Pathfinder is similar to dominator analysis [8], [4], but the underlying goals are different enough to make comparing them insufficient for our purposes. Dominator analysis simplifies a call graph by identifying basic blocks that always precede other blocks when evaluated from a global start node traversed to a global end node. However, PathFinder's analysis may start and terminate anywhere in the graph, with no node a dominator of any other, and cyclic paths are viewed as normal and desirable.

PathFinder is a proxy application, maintained as part of the Mantevo project [3] (www.mantevo.org). Unlike a benchmark, where rules constrain experimentation, proxy apps are designed for modification and experimentation, to any extent that remains relevant to an application developer. This enables,

for example, investigation of different programming models, languages, and mechanisms, existing, emerging, and future architectures, and even entirely new algorithmic approaches for achieving effective use of the computing environment within the context of complex application requirements.

II. PathFinder

A graph is defined as

$$G = \{V, E\},$$

for a set of vertices V and set of edges E. For a directed graph, the edges have a direction, e.g. $e_i \rightarrow e_j$, but $e_j \nrightarrow e_i$.

PathFinder searches directed graphs to find characteristic signatures, which is a specific sequence of labels that are associated with nodes in the graph. A path is a list of nodes where each node has an edge to the next node in the path. Paths that match a signature begin with a node labeled with the first label in the signature, and subsequent nodes in the path are associated with the remaining signature labels in order. However, many nodes may lie on the path between labels that are subsequent in the signature; the portion of a path between two subsequent labels in a signature is called a leg. The signature path terminates with a node associated with the last label in the signature. The set of nodes that make up a path between two labels is the path for a "leg" of the signature. The combination of all legs defines the complete signature path.

An example graph, illustrated in Figure 1, contains paths for

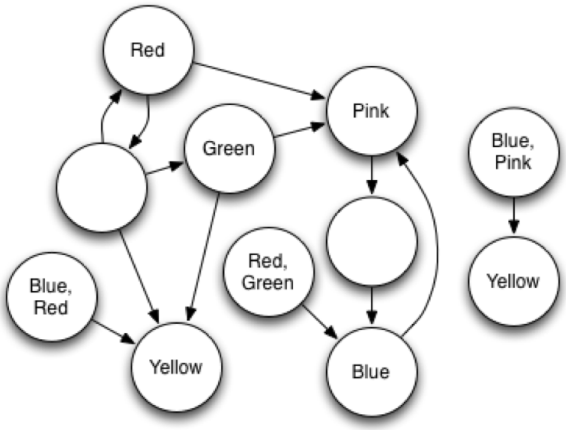

Fig. 1. PathFinder graph

the signatures `(Red, Blue)`, `(Red, Green, Blue)`, `(Red, Green, Yellow)` and `(Red, Red, Blue)` but not `(Red, Green, Blue, Yellow)`.

Not all nodes are labeled, some may have more than one label, and labels are not unique to any given node. The set of nodes that make up a path between two labels is the path for a "leg" of the signature. When all the legs are combined, that makes up the complete signature path.

Further, the graphs may be

- *noisy*. They may have missing edges, missing nodes, or entire sequences of nodes and edges that are constructed differently simply because of errors introduced in translating the binary into a graph. Although the graph algorithms mentioned above are well understood in the usual case, they must be modified to accept "fuzzy" matches to deal with noise in the graphs.
- *nested, directed, and potentially cyclic*. The exterior graph is the function call graph: nodes represent functions and edges represent calls from one function to another function. The interior graphs are control flow graphs (CFGs): each function node contains the CFG for that function, interior nodes represent basic blocks, and edges represent execution moving from one basic block to another.
  - Interior CFGs have edges to the exterior graph. When a basic block terminates by calling another function, the basic block has an edge to the destination function. Additionally, the containing function has an edge to the destination function (by definition).
  - The interior graphs are fairly localized, but the exterior graphs are not. Except for function calls, basic block nodes should transfer control only to other basic blocks in the same function. Similarly, function calls should transfer control to the one entrance basic block node that represents the beginning of the destination function.
  - The branching factors and structures of the two layers are very different. The exterior function call graph branching factor runs the gamut from very high to very low, and the exterior graph rarely has cycles of more than one node. (Recursive functions have one-node cycles.) The interior CFGs tend to have a branching factor of up to two, although jump tables cause rare branches of significantly more than two, and the interior graph frequently has cycles.
- *vastly different sizes*. Function call graphs may have from one to thousands of functions, and CFGs may have from one to thousands of basic blocks.
- *not complete*, as there are some subgraphs that are not connected with the remainder of the graph. Not all nodes are labeled, some may have more than one label, and labels are not unique to any given node. Any number of nodes may exist between any two labels in a signature, however loops are excluded. The set of nodes that make up a path between two labels is the path for a "leg" of the signature. When all the legs are combined, that makes up the complete signature path.
- *irregular with regard to metadata and signatures*, and potentially extremely complex. for example, nodes may have multiple labels[1], edges may have labels, labels may cross or encompass several nodes, and signatures may include combinations of labels or partially labeled subgraphs. PathFinder provides a simple example to begin addressing these types of problems.

PathFinder currently has two modes of operation. Normal operation searches for signatures as described above. Exhaustive mode determines whether or not a path exists between

[1]The current implementation of PathFinder only supports a single label.

each pair of labels in the graph. Plans include adding other modes.

A. Implementation

Because of the object density of the problem (graphs composed of nodes and edges, etc.), PathFinder has a distinct C++ flavor to its coding style. However, for a variety of reasons, PathFinder is written using the C programming language, with strong attention paid to clear software quality principles. There are many structures that have structure-specific functions associated with them, such as a "vector" storage pattern similar to `std::vector<Type>`. Without C++ templating, PathFinder has a lot of closely duplicated code. That said, most of PathFinder code is infrastructure and support code such as parsing or storage, but the heart of PathFinder is in its search algorithms.

B. Algorithm

PathFinder traverses the graph in a modified depth-first recursive search, comparing all adjacent nodes before recursing down the edges. if a match is found, the algorithm advances to the next label in the signature and recurses with the matched node as the new start. if there are no more labels in the signature the search is deemed a success, and the recursion is stopped. if no matches are found among the directly-connected nodes, the algorithm recurses along each edge with the current label still being the one being compared against. Once all edges have been checked without finding a match, then recursion is terminated as a failed search.

PathFinder maintains a list of nodes for each label represented in the graph. Each signature search begins at a node labeled with the first label in a signature. PathFinder checks adjacent nodes for the next label in the signature. if it is not found, PathFinder then does a recursive depth-first search until the next label is found. This completes the discovery of a leg of the signature. Once a leg is found, a new search is started for the next label in the signature. Loops are not allowed within a leg, but they may exist within the complete signature path. PathFinder iterates through all nodes with the first label until it finds a valid signature path or terminates proving that the graph does not contain the signature.

The description of the graph is defined in terms of a C structure, shown in Figure 2. The structs within it (`NodeType`, `Node`, `NodeList`, `interiorNodes`, and `EdgeList`) are essentially linked lists.

A pseudo-code sketch of the search algorithm is shown in Figure 3.

```
struct NodeStruct
{
char
   *label;
int
   id, labelIdx, nodeCount,
   edgeCount, entranceCount;

NodeType
   type;

Node
   *container; // If interior node, points to
               //  outer node containing this one.
NodeList
   *interiorNodes; // All nodes contained in
                   // this node's subgraph.
EdgeList
   *edges;  // Nodes that can be accessed from
            // this node (includes entrance nodes).
};
```

Fig. 2. Data structures

III. Performance Modeling, Results and Discussion

As PathFinder is one of the newest additions to the Mantevo suite of miniapps, understanding parameters that impact its performance is important. In this section we present performance characterization and cache behavior of the PathFinder miniapp. We begin by describing the setup environment and graph problem generation methodology used for this study, followed by performance characterization and impact of various parameters of the input graph on application performance. We conclude the discussion by presenting cache characteristics of PathFinder miniapp.

A. Experiment Environment

In this section, we describe the environment used for performing characterization studies for the PathFinder miniapp.

Machine Description:

We conducted experiments on Edison, a Cray XC30 system located at National Energy Research Scientific Computing Center (NERSC). Each compute node consists of dual socket Intel Xeon E5-2670 "Ivy Bridge" 12-core processors clocked at 2.4 GHz. Each core has a private 64KB L1 and 256KB L2 cache. All cores on a socket share a 30MB L3 cache. Each compute node uses 64GB DDR3 memory operating at 1600 MHz. The computing environment was set using module `PrgEnv-gnu/5.2.25`, which includes the gcc compiler version 4.9.1, with which we used flags `-static -fopenmp -O3`. Runtime profiling data was collected using the Cray Performance Analysis Tool (CrayPat) found in module perftool version 6.2.1.

Graph Generation:

The complexity of the graph problems are often determined by the number of edges and nodes in a graph. Similarly, for our PathFinder miniapp, the complexity of the signature search algorithm application is determined by the number of nodes in the graph, and characteristics of the signature searches, largely influenced by the number of unique labels in the graph and aggregated label count across all the nodes in the graph. The number of nodes determines the size of the graph, whereas the number of unique-labels and total-labels determine the

```
boolean FindRemainingSignatureFromCurrentNode  ( currentNode, signature, result, visited )
{
   if currentNode is in visited
      return FALSE // Cycle or a subgraph that has already been searched
   else
      push currentNode onto visited stack

   push currentNode onto result stack

   for each edge from currentNode
      if edge->targetNode->label == signature[0]
         if  signature[1] != NULL we have more legs in the search
            Create nextResult for next leg search
            Create nextVisited for next leg search
            success = FindRemainingSignatureFromCurrentNode
                        ( edge->targetNode, &signature[1], nextResult, nextVisited )
            if success
               result += nextResult
               delete nextVisited
               return TRUE // we've found the path
            else
               continue through edge checking
         else signature[1] == NULL
            push edge->targetNode onto result
            return TRUE // we've found the path
     else
        continue through edge checking
   // end of for each edge target compared against signature [0]

   If we've made it this far, none of the edges is a direct match to the current label; do a deeper search:

   for each edge from current node
      success = FindRemainingSignatureFromCurrentNode
                  ( edge->targetNode, &signature[0], result, visited )
      if success
         return TRUE, we've found the path
      else
         continue through edge searching loop

   If we've made it this far, we have no path:

   pop currentNode off of result // but not off of visited
   return FALSE;
}
```

Fig. 3. Search Algorithm pseudo code

signature search time. In order to study the impact of the total-labels in the graph on performance, we define *density* as the average number of labels per node in the graph. Therefore, the density of a graph problem is given by –

$$Density = \frac{Total\ labels\ in\ graph}{Number\ of\ nodes}$$

The PathFinder miniapp can operate in two different modes: *signature-search* mode, where a given signature is detected if present in the graph; or *exhaustive-search* mode, where a given path is determined if it exists between any pair of nodes for every label in the graph. Execution of the PathFinder miniapp can be divided into two phases: *build*, during which the program reads the input and builds a graph, an operation that is inherently *serial* in nature; and *search*, wherein a signature search is conducted. Signature searches performed in exhaustive-search mode are embarrassingly parallel, an individual path/leg of a signature can be treated as individual search and be distributed among available threads for computation. We use the exhaustive-search mode of operation for results presented in section III. This mode also mimics real-world scenarios wherein one is interested in finding the relationship between graph elements rather than focusing on a single search.

B. Performance Characterization

We begin by presenting execution time for various graph sizes. Figure 4 shows the average execution time for 1000-node, 2000-node and 4000-node graph problems. There are 800, 1600, 3200 unique-labels and 2000, 4000, 8000 total-labels in the graph for 1000, 2000 and 4000 nodes, respectively. We kept the ratio of unique-labels to number of nodes in the graph for this experiment constant at 0.8 and density or average number of labels per node to 2. These values are representative of real-world problems. However, in later sections we discuss the impact of change in the ratio of

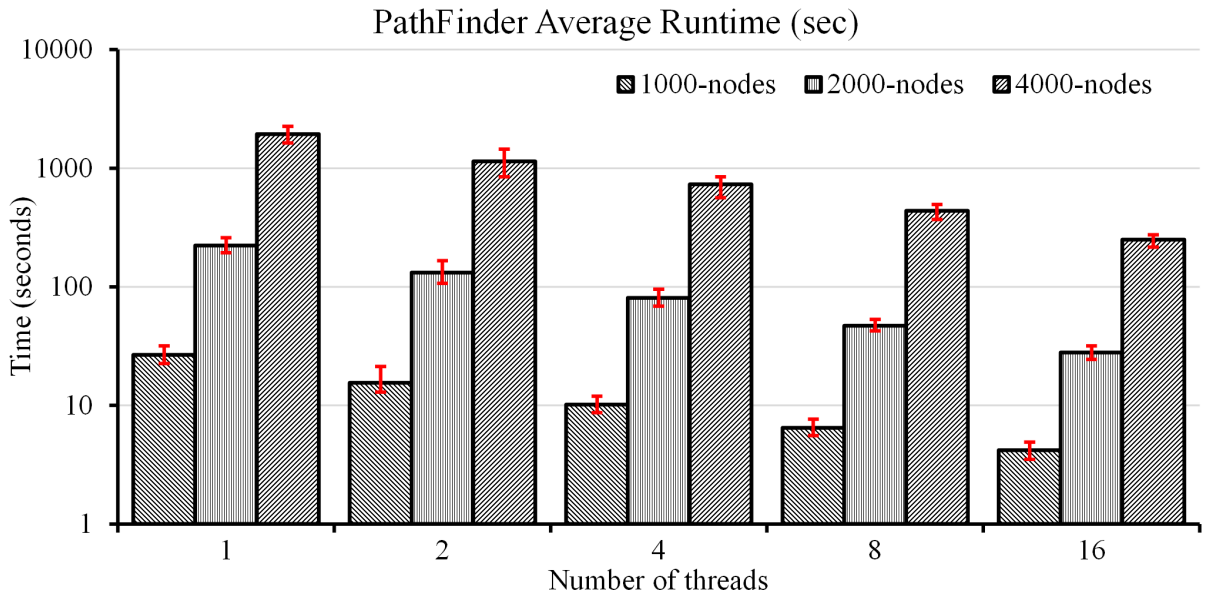

Fig. 4. Execution time for PathFinder miniapp

unique-labels to number of nodes and density of labels in the graph. For each problem size, we generate 10 different graph problems. Each of the 10 problem graphs are then executed and their performance characteristics captured. To capture the impact of inherent parallelism in the signature-search operation in the application, we use various numbers of threads for application execution. In Figure 4, the X-axis indicates the number of OpenMP threads used for execution, while the Y-axis indicates the average execution time in seconds on a logarithmic scale. Each column indicates the average of the execution time over the 10 graph problems. The variation bars on each column indicates the spread of the execution times for the 10 different problem runs between the min and max execution times.

From Figure 4, we observe that, with the same input parameters, execution time varies over 30% depending on the configuration of the graph. As problem size increases, the increase in runtime is exponential compared to the increase in problem size. Similarly, as the number of threads available for computation increases from 1 to 16, the average execution time for all problem sizes decrease, indicating that the search operation benefits from parallelism provided by additional threads. Even with additional threads, we observe average speedups of 1.7x, 2.7x, 4.4x and 7.3x for 2, 4, 8, 16 threads, respectively, indicating that we achieve nearly half the speedup per additional threads, and also the average speedup decreases with increase in threads available for computation.

We now look into how various input parameters of the graph affect the performance of the PathFinder miniapp.

We first discuss the impact of the number of nodes in the graph on execution time. Figure 5 shows how the execution time varies with increase in the graph size. The Y-axis shows the execution time on a $\log_2$ scale. We present results for

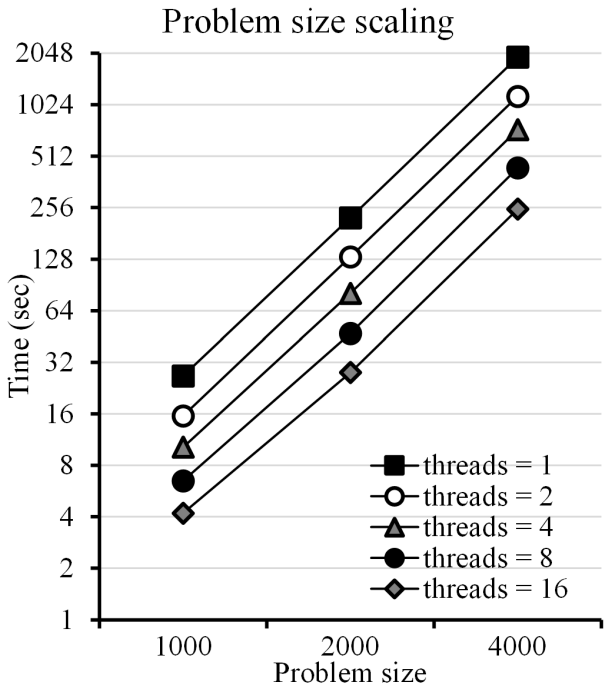

Fig. 5. Execution time vs. problem size scaling for PathFinder

various number of threads used for application execution. Again, the density was kept constant at 2 and the ratio of unique-labels to number of nodes in the graph at 0.8. From the plot, we observe that for various threads, as the problem size grows, the increase in execution time is exponential, indicating that the amount of computation increases exponentially with problem size.

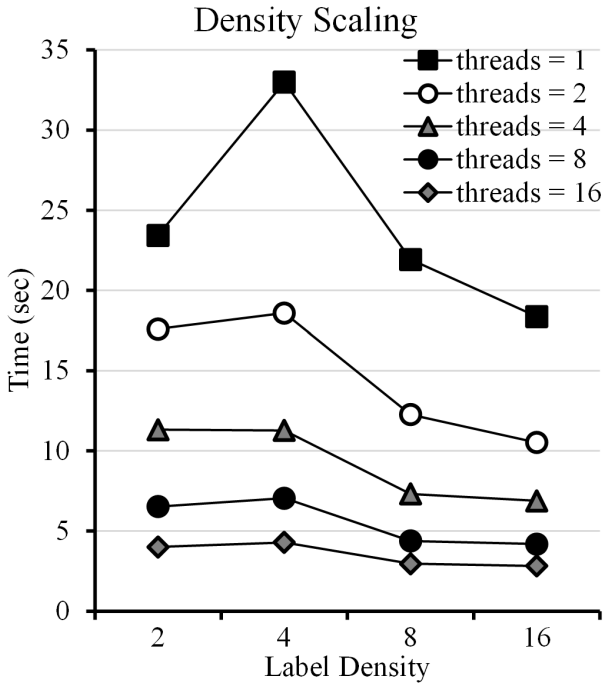

Fig. 6. Density scaling for PathFinder miniapp

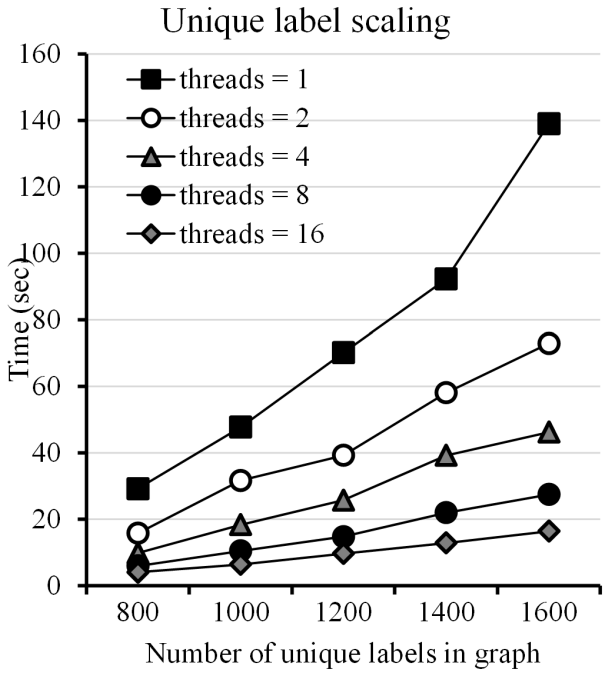

Fig. 7. Unique-label scaling for PathFinder miniapp

Figure 6 shows the impact of increase in density (average number of labels per node) of the graph on performance (execution time). For this experiment, we consider a 1000-node, 800-unique-labels graph and we vary density from 2 to 16. We plot results for execution time for various numbers of threads used for computation. From Figure 6 we observe that in every thread execution scenario, the average execution time decreases as the density of the graph increases. Considering that we are running PathFinder in exhaustive-search mode, the goal is to find a path between a pair of nodes containing the searched label-pair. Therefore, as density increases, there is higher probability of finding a node traversal containing the label early in the graph traversal. This expedites an individual label-pair search through the graph thereby decreasing execution time. Also, we observe that the decrease in runtime with increase in density is highest in 2-thread execution; this can be attributed to 2-thread execution being a good fit for this problem size, and we expect that as problem size increases, higher numbers of threads will be beneficial to execution.

Figure 7 plots the impact of increase in the unique-labels in the graph on the performance. For this experiment we consider 1000-node, 4000-total-labels graphs, and we vary the number of unique-labels between 800-1600. We plot results for various numbers of threads allowed to be used for computation. In this experiment with a constant number of nodes and total-labels in the graph and executing in exhaustive search mode, the memory footprint of the graph remains constant; only the search space increases with higher unique-label count. The total number of label-pair searches increase from around 800^2 to around 1600^2 for unique-labels increasing from 800 to 1600. We observe that as the number of unique-labels increase, the runtime increases. Increase in runtime is higher for runs with lower numbers of available threads for computation. For 16-thread execution, runtime increased by 4X while it increased by over 5X in single-thread execution. Increase in the runtime can be directly correlated to the increased number of searches in the application execution.

The performance characterization results presented previously were for test-case sized graph problems. For real-world problems, much larger graph problems are expected. To that extent, we now examine how the PathFinder miniapp scales under strong scaling. For this experiment we consider 8000-nodes, 6400 unique-labels and 16000 total-label graph problems. For strong scaling, we analyzed the application execution running between 1-16 threads. Figure 8 shows the strong scaling results for the PathFinder miniapp. In addition to the actual execution results, we plot an *ideal scaling* scenario wherein for each number of threads used for computation, we estimate ideal execution time by dividing the single-thread execution-time by the number of threads used during a particular computation. From the graph, we observe that the PathFinder miniapp scales well with increases in the computation resources. As the number of threads used for execution increases, the execution time decreases and tracks the *ideal-time* curve albeit with a fixed offset. The fixed offset is attributed to the serial nature of the graph build phase as well as the overhead of parallelization for distributing work among increasing numbers of threads. This offset decreases as the number of threads available for computation increases, indicating good strong scaling characteristics.

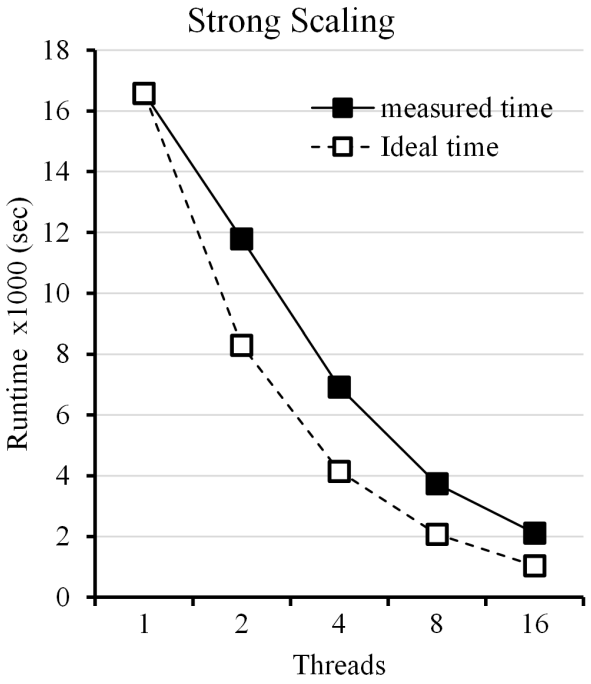

Fig. 8. Strong scaling for PathFinder miniapp

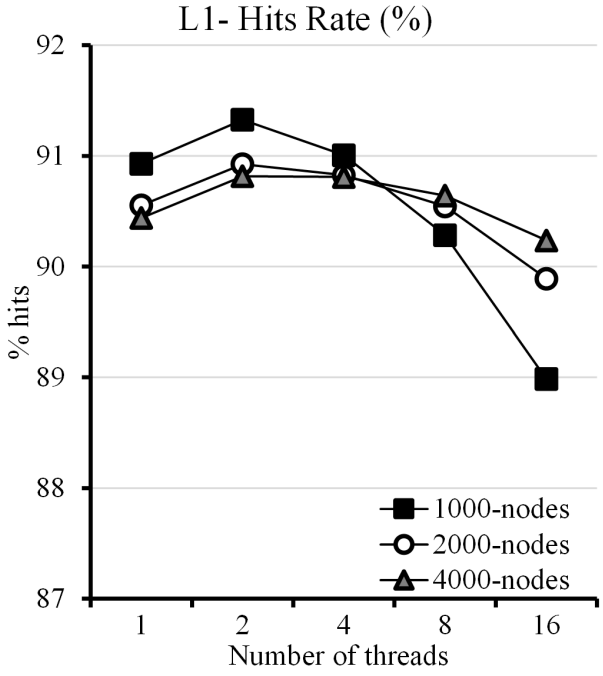

Fig. 9. L1 cache Hit Rate for PathFinder miniapp

With PathFinder being a graph-based miniapp, the complexity of the search operation increases exponentially with increase in the graph size, and hence it becomes difficult to perform weak-scaling experiments, wherein one increases the problem size with increase in the resources. At this point we are working on correlating problem size with the complexity of the graph and analyzing its impact on performance. We leave this task of characterizing weak scaling for PathFinder as future work.

C. Cache Characteristics

The behavior and utilization of the memory hierarchy, especially the cache hierarchy, have a strong impact on the performance of any application. Understanding these characteristics is important to application developers, as it often indicates where improvements can be made to maximize performance of applications. It becomes especially imperative that one understands this behavior for new graph processing applications, as their cache utilization behavior is significantly different from other CSE (Computational Scientific and Engineering) applications. Figure 9 presents L1 cache behavior of the PathFinder miniapp. The Y-axis in Figure 9 indicates the percentage of L1 cache accesses which hit in the L1 cache, while the X-axis represents various numbers of threads used for computation. We consider three different problem sizes consisting of 1000-node, 2000-node and 4000-node graph problems with *0.8*nodes* unique-labels and *2*nodes* total-labels in the problem graph. For each problem size we generate 10 random graph problems, and the average cache hit rate across all graph problems is presented here. For all three problem sizes, we observe a L1 cache hit ratio between 89-92%. The majority of execution runs across various numbers of threads have L1 cache hit rates in excess of 90%. Although this indicates significant L1 cache locality, these hit rates are lower than those observed in other CSE applications where L1 cache hit rates are observed to be $\geq 95\%$. As the number of threads used for computation increases, for every problem size, we see a decrease (albeit minor) in L1 cache hit rates. However L1 cache hit rate remains nearly constant irrespective of problem size.

In Figure 10 we present the L2 cache hit rate for the PathFinder miniapp. We use the same problem sizes that were described above for gathering L1 cache statistics. As seen from the graph, the L2 hit rate is around 60% for 1000-node, 47% for 2000-node and 40% for 4000-node graph problems respectively. As problem size increases, the L2 hit rate decreases. This behavior results because when the problem size increases, a longer graph traversal is required during every search operation. With graph traversals in a larger graph spanning a larger memory footprint, the pointer chasing is spread out across a larger memory, leading to poor spatial locality during consecutive memory accesses. Scenarios such as a miss in the L1 cache causing a miss in the L2 cache are more likely to occur, leading to node information to be fetched from L3/ Last Level Cache (LLC) or memory. This L2 cache behavior is unique to PathFinder and very different as compared to other CSE applications. With increases in the number of threads for computation, we observe slight increases in L2 cache hit rate, which can be attributed to a larger combined L2 cache available across all the threads to the

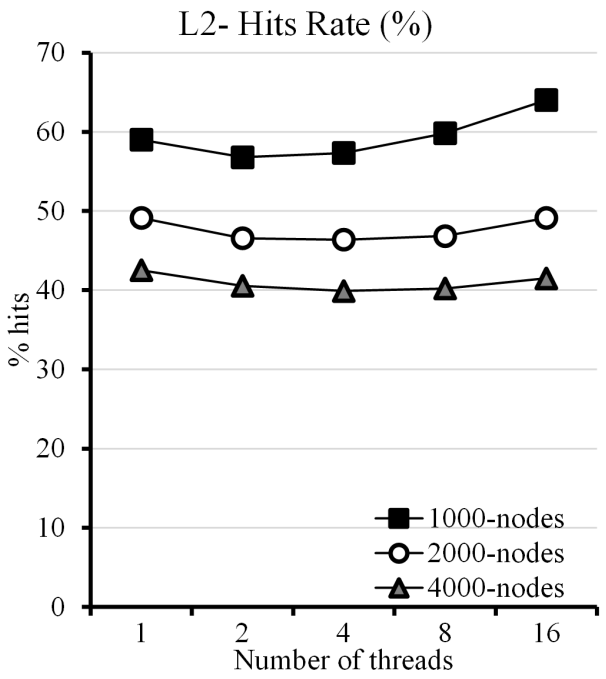

Fig. 10. L2 cahe Hit Rate for PathFinder miniapp

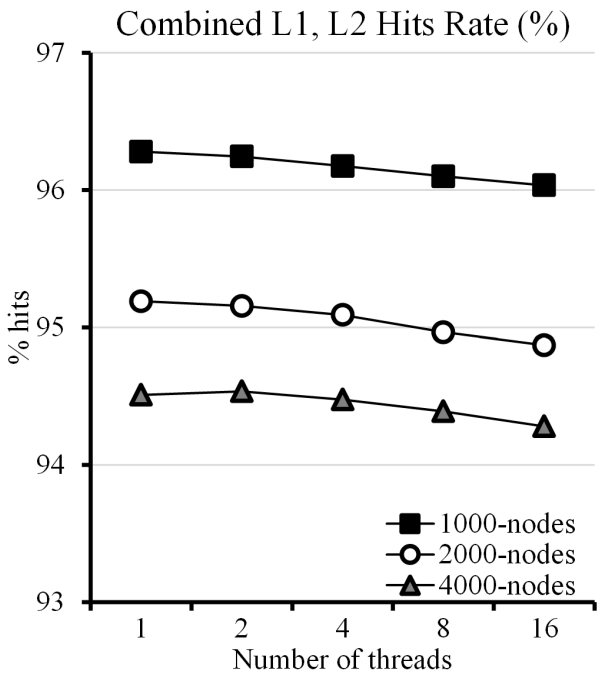

Fig. 11. Combined private cache Hit Ratio for PathFinder miniapp

program. Even with the increase in number of threads leading to more L2 cache being available to the program, the inherent nature of graph search means that no individual L2 cache can hold the complete problem memory footprint, which also leads to lower L2 cache hit ratios. Further cache studies are needed to determine which cache architectures/ organizations would lead to better L2 cache utilization. The micro-architecture evaluation of the cache architecture is beyond the scope of this paper, though we will be addressing this topic in future work.

In Figure 11 we show the combined hit rate for the private caches available to each core. The Y-axis of Figure 11 shows the combined L1 and L2 cache hit rate percentage, while the X-axis represents the number of threads used for computation. We observe that the combined hit rate is over 94% for all problem sizes. The combined hit rate is higher than the individual hit rates observed for L1 or L2 cache indicating that the presence of L2 cache improves overall hit rates. Given that L1 cache accesses dominate aggregated L1 and L2 cache accesses, we observe that the combined hit-rate tracks L1 cache hit rate more closely. Similar to the L1 cache behavior, when either the problem size or the available threads for computation increases, the combined cache hit rate decreases.

We also measured the L3 cache hit ratio and observed nearly a 100% hit rate for various problem sizes. The L3 cache hit-rate drops to around 98% for all problem sizes when executing with 16 threads. The L3 cache being the Last Level Cache (LLC) is shared among all the available threads. As the number of working threads increases, the average L3 cache size available to each thread decreases. Even though IvyBridge processors have a large shared L3/LLC cache, with 16 threads and assuming uniform sharing, each thread is limited to around 2.5MB of the L3 cache on the socket. This leads to an increase in the number of L3 cache misses, thereby decreasing the observed hit rate. We also measured the memory footprint for various problem sizes running with various numbers of threads for execution. The memory footprint is shown in table I. From Table I we observe that the memory footprint increases with the number of threads used for computation. The memory footprint of the application increases from $\approx$ 23MB to $\approx$ 201MB for 1000-node graphs. The memory footprint remains nearly constant with the increase in the graph problem size.

TABLE I
MEMORY FOOTPRINT (MB) FOR PATHFINDER MINIAPP

Number of Threads	Graph problem size (# nodes)		
	1000	2000	4000
1	23	25	29
2	40	42	45
4	72	74	77
8	136	139	142
16	201	204	207

To summarize, in this section we presented performance characterization of the PathFinder miniapp with different problem sizes and various numbers of threads. We showed the impact of increase in the number of nodes, unique-labels and

total-labels in the graph on the performance of the application. We also demonstrated and characterized scaling properties of the PathFinder miniapp. With cache characteristics and behavior having a strong impact on the performance of any application, we presented the cache characteristics of the PathFinder miniapp. Both L1 and L3 caches have a cache hit ratio of over 90% whereas the L2 cache hit ratio is substantially lower (less than 60%). Also the L2 cache hit ratio decreases as the problem size increases, suggesting, poor utilization of L2 cache in this graph-processing miniapp. This type of behavior merits further attention, as it differs from that of other CSE applications.

ACKNOWLEDGMENT

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

IV. SUMMARY AND FUTURE WORK

We exercised PathFinder with different problem sizes and various numbers of threads. We showed the impact of increases in the number of nodes, unique-labels and total-labels in the graph on the performance of the application. We also demonstrated scaling properties of the PathFinder miniapp. With cache characteristics and behavior having a strong impact on the performance of any application, we showed the cache characteristics of the PathFinder miniapp. Both L1 and L3 caches have a cache hit ratio of over 90% whereas the L2 cache hit ratio is substantially lower. Also the L2 cache hit ratio decreases as the problem size increases, suggesting poor utilization of L2 cache in this graph-processing miniapp.

This initial implementation of PathFinder serves as a starting point for these sorts of signature searching explorations. Capabilities are being added that will provide stronger and deeper means for statically analyzing executables in order to characterize more complex and potentially nebulous control flows. Additional capability is being considered for addition to PathFinder. For example, the current implementation traces calls but does not analyze the path of returns. This would provide stronger context for the call trace. We continue to prepare for significantly larger problem sets, which would require more complex techniques for internode path finding.

These algorithmic capabilities will require increasingly more powerful and complex computing capabilities. Therefore, a major focus of our work is on emerging and expected future architectures. In particular we are preparing for many core nodes, with large hierarchical stack memory systems, perhaps with applicable logic assistance (e.g. forward referencing), with node interconnects providing significantly increased injection rates and bandwidth, but with proportionally less global bandwidth. System software is being developed that provides strong support for the sort of task parallelism inherent in PathFinder and other such algorithms, with integrated internode data movement protocols [9]. In particular, the constraints imposed by the traditional bulk synchronous parallel programming model (BSP [10]) will be significantly relaxed. A deeper discussion of this and other associated capabilities are described in [1].

This style of computing system will inspire a re-thinking of the sorts of problems that could be addressed by this integrated capability. For example, we envision operating on larger and multiple binaries simultaneously, requiring distributed memory computing environments.

ACKNOWLEDGEMENTS

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

[1] J. Ang et al. Abstract Machine Models and Proxy Architectures for Exascale Computing. In *Proc. of the First International Workshop on Hardware-Software Co-Design for High Performance Computing (Co-HPC)*, 2014.

[2] D.A. Bader and K. Madduri. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. In *Proc. 12th International Conference on High Performance Computing (HiPC 2005)*, volume 3769, pages 465–476. Springer-Verlag Berlin Heidelberg, December 2005.

[3] R.F. Barrett, P.S. Crozier, M.A. Heroux, P.T. Lin, H.K. Thornquist, T.G. Trucano, and C.T. Vaughan. Assessing the Validity of the Role of Mini-Applications in Predicting Key Performance Characteristics of Scientific and Engineering Applications. *Jounral of Parallel and Distributed Computing*, 2014. To appear.

[4] Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. A practical interprocedural dominance algorithm. *ACM Trans. Program. Lang. Syst.*, 29(4), August 2007.

[5] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, and U.V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.

[6] L.C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[7] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1), December 1998.

[8] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), New York, NY, USA, 1959. ACM.

[9] D.T. Stark, R.F. Barrett, R.E. Grant, S.L. Olivier, K.T. Pedretti, and C.T. Vaughan. A Dynamic Runtime with Co-Scheduling of Work and Communication Tasks for Hybrid MPI+X Applications. In *Workshop on Exascale MPI (ExaMPI)*, 2014.

[10] L.G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33:103–111, August 1990.