World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:15, No:3, 2021

# Analyzing the Factors that Cause Parallel Performance Degradation in Parallel Graph-Based Computations Using Graph500

Mustafa Elfituri, Jonathan Cook

***Abstract*—**Recently, graph-based computations have become more important in large-scale scientific computing as they can provide a methodology to model many types of relations between independent objects. They are being actively used in fields as varied as biology, social networks, cybersecurity, and computer networks. At the same time, graph problems have some properties such as irregularity and poor locality that make their performance different than regular applications performance. Therefore, parallelizing graph algorithms is a hard and challenging task. Initial evidence is that standard computer architectures do not perform very well on graph algorithms. Little is known exactly what causes this. The Graph500 benchmark is a representative application for parallel graph-based computations, which have highly irregular data access and are driven more by traversing connected data than by computation. In this paper, we present results from analyzing the performance of various example implementations of Graph500, including a shared memory (OpenMP) version, a distributed (MPI) version, and a hybrid version. We measured and analyzed all the factors that affect its performance in order to identify possible changes that would improve its performance. Results are discussed in relation to what factors contribute to performance degradation.

***Keywords*—**Graph computation, Graph500 benchmark, parallel architectures, parallel programming, workload characterization.

## I. Introduction

IT has long been known that different types of computations can require very different approaches to their parallelization and that some computations are much easier to parallelize than others. In particular, irregular computations, such as those performed over large graph data structures, typically exhibit poor speedup relative to the resources available [1], [19]. The Graph500 benchmark [2] was created to be a representative application for evaluating system performance on such applications.

Graph500 specifies a set of graph computations that must be performed, allowing for custom implementations of the computations, but it also includes several example implementations, from a shared memory implementation to distributed and combination versions. We performed an in-depth analysis of three of these representative implementations in order to get a better understanding of where the loss of speedup is generally being produced and to see if there might be algorithmic or architectural changes that might help produce more scalable parallel graph computations. While much work has been done at the overall net level of parallel performance measurement and improvement for graph algorithms (see Section VI), much less work has been done in delving deep into the individual causes of the observed parallel performance. So even though much work shows how particular algorithmic or platform changes improve (or harm) the overall performance of some graph computation, less is known about individual contributions to the overall performance.

This paper contributes an investigation into what factors are most significant when it comes to explaining the causes of poor speedup in the Graph500 parallel computations. It contributes an assessment of factors for shared-memory, distributed, and hybrid versions of Graph500. The nature of this paper is a short summary of much data that have been gathered; for a full presentation in the breadth of the investigation, please refer to [3].

Section II details the applications we used in this work; Section III presents results from evaluating the shared-memory version of Graph500, Section IV evaluates the distributed version, and Section V evaluates the hybrid version. Finally, Section VI presents related work, and Section VII presents future work and conclusions.

## II. Platforms and Experimental Setup

Graph500 is a two-kernel parallel benchmark that has an undirected graph generator as a first code kernel, and then a parallel breadth-first search (BFS) over the graph as the second. The BFS algorithm requires no locking as it does not perform potential conflicting updates. In this BFS kernel, 64 separate BFS searches are performed, each from a randomly selected starting node. Although the graph generation is parallelized, it is the BFS kernel that is the heart of the Graph500 benchmark and that consumes the vast majority of execution time. Performance for Graph500 is measured in TEPS or traversed edges per second. Graph500 has essentially no computation associated with the visiting of each node, other than the finding of its edges and the continued BFS search. The problem size is denoted by an integer scale size; the problem graph then has $2^{scale}$ number of nodes in it.

Our experiments have been performed over a period of time and have utilized three different hardware platforms, although all of them are essentially similar. Most of the evaluation of pure shared memory parallelism was performed on an Intel Xeon platform with two 10-core CPUs, and with each core

M. Elfituri is with SUNY Morrisville, Morrisville, NY 13408 USA (e-mail: elfituma@morrisville.edu ).

J. Cook is with New Mexico State University, NM 88003 USA (e-mail: jcook@nmsu.edu).

being 2-way hyperthreaded. Some of the distributed parallelism evaluation was performed on an older 24-node cluster with 2 quad-core Xeon 5335 processors and 12 GB of RAM on each node and an Infiniband interconnect. Some of the distributed and all of the hybrid evaluation was performed on a newer small 10-node cluster with 24 cores per node (12 cores per CPU), configured with hyperthreading for 48 hardware hyperthreads per node, and 256GB of RAM; it also has an Infiniband interconnect.

Problem sizes were from scale 20 to scale 28, depending on the platform used, the resources available, and the time constraints for which the platform could be used under. The shared memory evaluations, running only on one node, obviously used the smaller scales. We generally tried to use the largest scale that would fit on the resource configuration, though for some evaluation we scaled back based on time concerns. We did not see significant changes in overall numbers as we changed scales, and thus do not view the problem scale as an important parameter, except at extremes. All implementations were built with Gnu compilers, and we used OpenMPI for the MPI implementation. All systems use some variant of Linux, with the new cluster running CentOS. Most data reported in this paper are averages of five runs.

## III. Shared Memory Evaluation

In this section, we present results from evaluating the pure OpenMP version of the Graph500 benchmark. We will refer to this implementation as Graph500-OMP. We used a tool we created, PGOMP [4], that uses library interposition to provide a detailed inspection of OpenMP overheads for the Gnu OpenMP implementation, and is similar to tools provided with commercial OpenMP compilers

### *A. Sequential Time Variation*

Amdahl's Law suggests calculating ideal speedup from the fraction of time spent in the sequential code versus the time spent in parallel code. The difference in observed speedup from this ideal speedup is then due to the various overheads incurred (e.g., communication, synchronization). For shared memory applications, however, simply measuring the sequential portion on a one or even two-thread configuration may not give an accurate assessment of the true sequential portion cost. This is because each thread configuration affects the shared resources and their state after a parallel section is completed, and this can affect the performance of the sequential portion.

Our first measurement, then, is the time spent in the sequential portion. While the overall portion of sequential time for Graph500-OMP is small, we observed variations from about 0.5% of total single-threaded time to about 1.5% of total single-threaded time; this may seem inconsequential, but it results in large effects. Fig. 1 shows the effect that these varying sequential execution times have on what Amdahl's Law would offer as an upper bound on speedup. Each line shows the curve from Amdahl's Law based on the measured sequential execution time for the number of processors indicated. Graph500-OMP has significant variation in sequential execution times, and the figures show that just this variation can explain a significant amount of the speedup loss that the actual executions suffer.

In these experiments, the sequential time observed for Graph500-OMP with one thread was 0.74%, while over the various thread configurations the maximum observed was 1.78%. This difference will significantly explain part of the observed performance loss from what Amdahl's Law would predict, and ignoring it would lead to erroneous results. This observation could be very important in the future where processors may have many, many cores available. In such a case, even using Amdahl's Law for rough upper-bound estimates could be very wrong in the many-core case. Amdahl's Law is simply invalid for the context-sensitive execution performance of modern processors and memory hierarchies; nothing is constant across configuration changes.

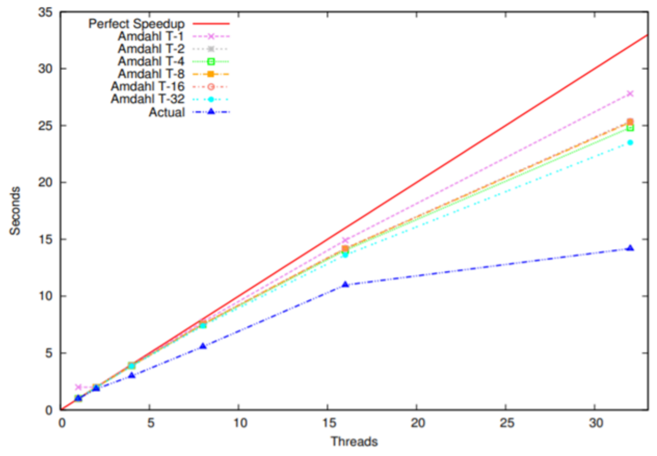

Fig. 1 Graph500-OMP ideal speedup variation for observed sequential time variation, along with actual speedup observed

### *B. Parallel Overhead*

*a) Library Overhead*: Library routines, other than lock orbarrier waiting (which are counted elsewhere), run too fast for good timing measurements, so we also used PAPI along with PGOMP to obtain an instruction count for the OpenMP library code. These instruction counts were consistently many orders of magnitude smaller than application instruction counts, and were also consistent across thread configurations, and so we conclude that library overhead is negligible.

*b) Lock and Critical Section Overhead:* We measured the time spent in OpenMP locks and critical section constructs, but because Graph500-OMP is essentially read-only computation, these are negligible (and actually measured as 0). We measured these factors in other applications to test that our framework correctly captures them: for example, the SSCA2 graph benchmark [24] was measured to have 0.64% overhead spent in critical section waiting.

*c) Barrier Overhead*: OpenMP uses a barrier construct at the end of a parallel loop and this synchronization time is significant for Graph500-OMP. Table I shows barriers accounting for up to almost 30% of the overhead.

TABLE I
OVERALL FACTOR CONTRIBUTION TO PERFORMANCE LOSS

| Threads | Measured Speedup | Ideal Speedup (Amdahl) | Sequential Variation Overhead % | Barrier Accounted Overhead % | Lock/CS Accounted Overhead % | Memory Hierarchy Overhead % |
|---|---|---|---|---|---|---|
| 1 | 1.00 | 1.00 | - | - | - | - |
| 2 | 1.84 | 1.98 | 1.58 | 9.38 | 0.00 | 88.04 |
| 4 | 2.99 | 3.89 | 3.33 | 21.28 | 0.00 | 75.93 |
| 8 | 5.56 | 7.54 | 5.86 | 21.53 | 0.00 | 72.61 |
| 16 | 10.98 | 14.20 | 12.27 | 17.31 | 0.00 | 70.42 |
| 32 | 14.19 | 23.51 | 15.82 | 18.95 | 0.00 | 65.32 |

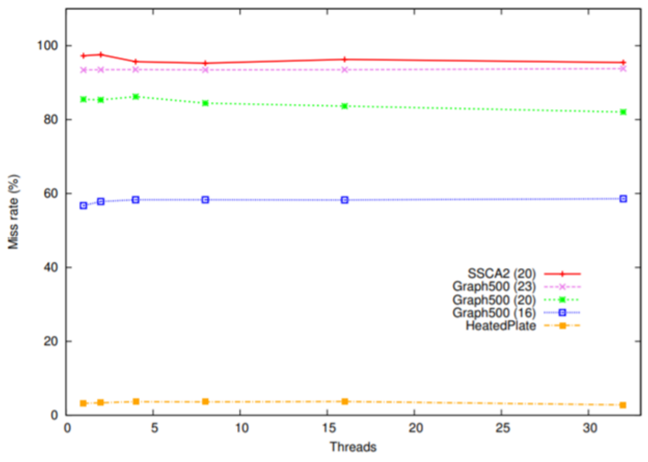

Fig. 2 Graph500-OMP L2 data cache miss rate for various scales. Also shown for comparison are SSCA2 and a simple heat transfer

d) *Memory Overhead*: The other potential factor inexecution time variation is the local memory hierarchy performance. Fig. 2 shows the level 2 data cache miss rates for three different problem sizes for Graph500-OMP, along with two other applications for comparison1. This indicates that the local cache performance of the parallel computations is not significantly varying; as far as the cache is concerned, their computation profile is consistent. Thus, we can assume that changing local cache performance is not a factor in any poor speedup.

We also measured instruction counts in threads to be sure that no algorithmic or architectural anomalies are occurring. We conclude that based on the instruction counts and the consistent memory performance, the parallelized sections execute at essentially the same rate in each of the N-processor configurations. There is no significant variation that might help explain the poor speedup.

## C. Results

For the portion of overhead due to the memory hierarchy, other than on simulation it is virtually impossible to directly measure this. Counters do not give time measurements, and most CPUs do not have coherency counters (and none we have to do). Thus the overhead accounted for by the memory hierarchy must be what is leftover from how much is accounted for by the factors that we did measure.

Table I shows the concluding results from putting together all of the measurements. Measurements are mostly taken from scale 20 for Graph500-OMP. The last four columns are the percentage of contribution to performance loss from the varying sequential execution time, the barrier synchronization time, the lock or critical section blocking time, and then finally the memory hierarchy contribution, which is calculated as the rest of whatever is left from the other three. Recall that measurements led us to conclude that there was virtually no contribution from varying parallel execution performance nor from the parallelization management code in the OpenMP framework library.

For Graph500-OMP, the sequential time variation is a significant factor, even increasing in significance as the thread count goes higher; if there is reasonably large sequential variation, Amdahl’s Law implies that it will get more significant as concurrency increases. Barrier overhead, for all configurations other than the 32-thread one, remains high. We surmise that Graph500-OMP has consistently high workload variation between the threads; this could potentially be a point for improvement.

The most important point of the Graph500-OMP results is that if we had assumed that the sequential execution time part was constant, as Amdahl’s Law does, we would have attributed much more overhead to the memory hierarchy, and would believe that we could achieve more improvements that might truly be possible (without addressing the sequential part). The differing sequential times here result in up to a 15% difference in the ideal parallel performance, which is insignificant.

The next step for this analysis would be to run Graph500-OMP under a memory simulator and try to break down the cost into per-core hierarchy costs and coherency costs. Since shared-memory parallelism captures its communication cost in the implicit memory hierarchy operations, detailing what happens here is the important next step. Many ideas are being pursued in regards to more efficiently connecting data and computation in these types of irregularly parallel, data-intensive applications, from active messages to radical processor-in-memory architectures. What is clear is that on present architectures, data access is a large portion, though not all, of the cause of poor speedup in Graph500-OMP.

[1] SSCA2 is another shared-memory graph benchmark, and HeatedPlate is a simple heat transfer computation

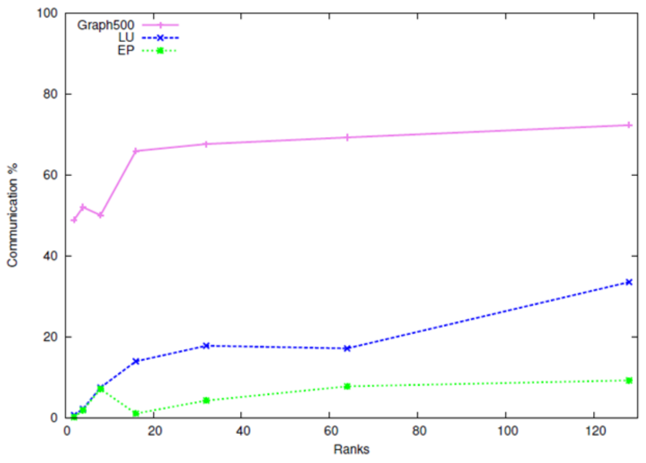

Fig. 3 Graph500-MPI MPI overhead. Total application time and time in MPI routines are shown

## IV. Distributed Evaluation

In this section, we present results from evaluating the pure MPI version of the Graph500 benchmark. We will refer to this implementation as Graph500-MPI. For these results, we mostly used the mpiP tool [5], which gives time breakdowns of all of the MPI calls that an application performs. For this version we did not measure sequential time variation; because this version uses pure MPI without local threading, there is no ending of local parallel sections and beginning of sequential sections; processes may wait at a synchronization point, but this will be measured elsewhere. Indeed when inspecting the implementation code, there is very little synchronization. It uses asynchronous sends and receives for most of the communication, using an all-reduce only at the end of a computation.

Fig. 3 shows the total overhead measured for Graph500-MPI (and compared to two NASA Parallel Benchmarks, EP and LU). Graph500-MPI obviously spends a large portion of its time in MPI—in most configurations, about 70%.

Fig. 4 shows the main components of the time spent in MPI. The dominant factor of MPI time is the routine MPI Test, which simply is a local test to see if some asynchronous communication has been completed. This is consistent with Graph500-MPI's use of asynchronous communication and seems to indicate that Graph500-MPI is mostly bound by waiting on data. Given that the sends and receives are asynchronous, their direct overhead is going to be mostly unavoidable, unless the data packaging and amount could somehow be optimized. Thus in this section, we will focus on the time spent waiting (e.g., in MPI Test). The mpiP tool breaks down MPI routine overhead based on call sites, and so we can learn more from the large portion of overhead due to MPI Test.

The Graph500-MPI code is organized with a large macro that has an initial loop that checks for received data messages using MPI Test, and then in a second loop uses MPI Test again to check for sending buffers that have emptied because of a completed send. The macro itself is used in two (prominent) places, one at the beginning of each work iteration, and one where it needs to send data to another process and is waiting for the send buffer to become available. These four separate call sites of MPI Test and their proportion of overhead are shown in Table II, labeled with the first part denoting the macro use site (top of work iteration or waiting for send buffer) and the second part the place in the macro itself (checking for received messages or checking for empty send buffers).

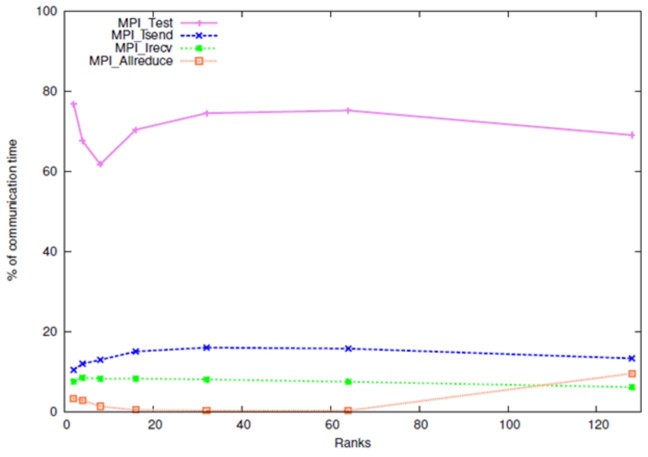

Fig. 4 Graph500-MPI overhead breakdown for the four largest components

TABLE II
MPI Test Call Site Overhead Breakdown.

| Call Site | App % Overhead | MPI % Overhead |
|---|---|---|
| Top-Recv | 18.48 | 25.58 |
| Send-Recv | 13.56 | 18.89 |
| Send-Send | 9.45 | 13.08 |
| Top-Send | 1.49 | 2.06 |

We hand instrumented the macro to collect detailed performance data on how MPI Test was being used. Fig. 5 shows a detailed look at how MPI Test behaves in Graph500-MPI. The layers in the graphs are histogram buckets for call durations, from 0 to 3 seconds. The buckets stack for a total number of calls. The X-axes are the MPI processes (ranks) and the algorithm iteration (recall that Graph500-MPI performs 64 spanning-tree constructions, choosing a random starting vertex each time). Two things should be taken away from these graphs. One is that the top graph shows that processes have an equitable distribution of the waiting (testing) time—there are no large distinctions between the ranks. Two, the bottom graph shows that most algorithm iterations are consistent with each other, but a few (13 in this graph) clearly spike the total amount of waiting time. We interpret this to be that for most searches, the graph partition is suitable and produces consistent results, but some starting vertices cause the search to perform quite poorly. We do not offer an opinion as to whether it would be good to try to improve the performance of these few searches. Notably, there are not any large dips below what seems to be a sort of baseline cost to doing a BFS, so no starting node causes some large increase in BFS performance.

The Graph500-MPI code uses one sending buffer for every other rank; if a rank needs to send a message while the previous asynchronous send to the same rank is still active, it needs to wait. A natural investigation into this cost would be to allocate two buffers for every other rank. We made this small change, with the results shown in Fig. 6. There is a reasonably significant improvement in speedup, but not to the amount that approaches removing the large waiting costs. For example, in one set of runs, MPI Test goes from 69% of total application time to 63% of the total time when two buffers are used.

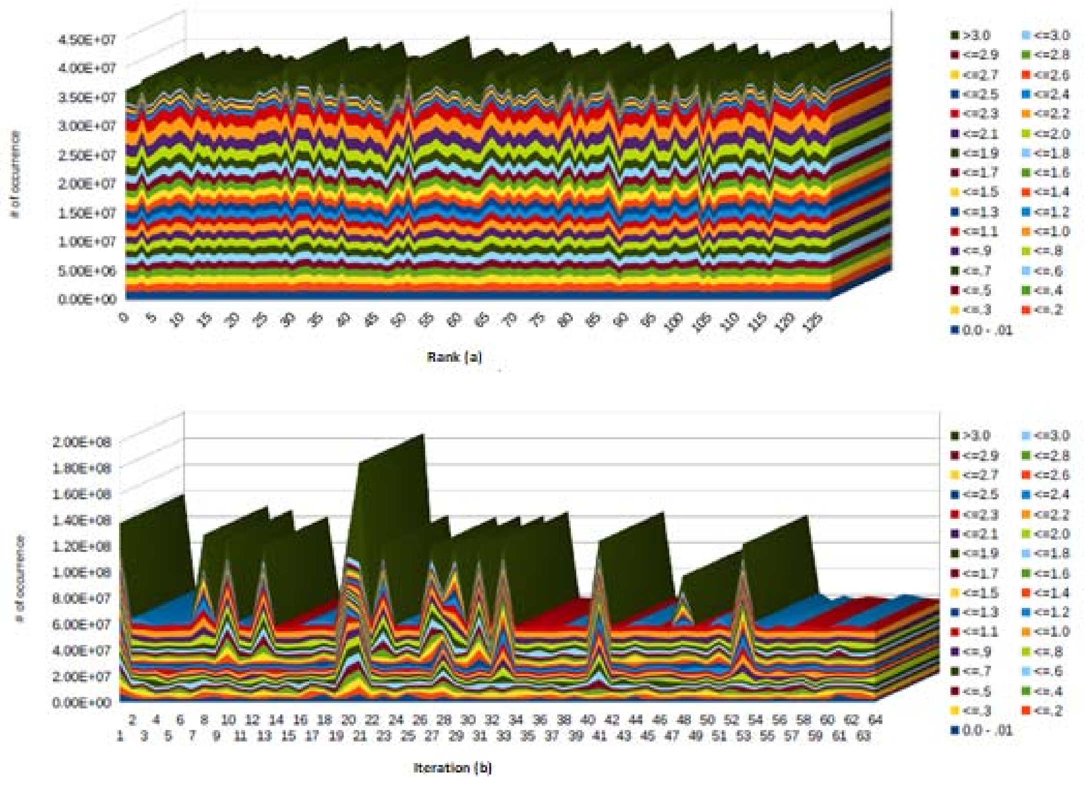

Fig. 5 MPI Test execution time histogram (layers) by rank (a) and by algorithm iteration (b)

TABLE III
PERFORMANCE USING ONE AND TWO MACROS

| | MPI % | | Performance | |
|---|---|---|---|---|
| Number of Macros | MPI_Test operations | Total | TEPS | Speedup |
| One Macro | 67.96 | 72.22 | 2.44E+08 | 37.84 |
| Two Macros | 67.65 | 72.90 | 2.36E+08 | 36.56 |

Another useful investigation to check is whether the coupling of the two separate tests in the macro causes some of the inefficiencies. To this end, we split the macro into the receiving and sending parts, and then only use each at the appropriate points in the main computation loop. Table III shows a sample result of this experiment, which resulted in essentially no change in the amount of waiting overhead. We conclude that the waiting is not dependent on this aspect of the structure of the Graph500-MPI implementation but rather is more inherent to the nature of the computational problem.

In summary, Graph500-MPI has most of its overhead in waiting for data to receive or to be sent, and seems to run out of work and is communication bound. Some improvement could be made, as shown with the dual send-buffer experiment, but large improvements need a different computational and communication model (as will be seen in the next section).

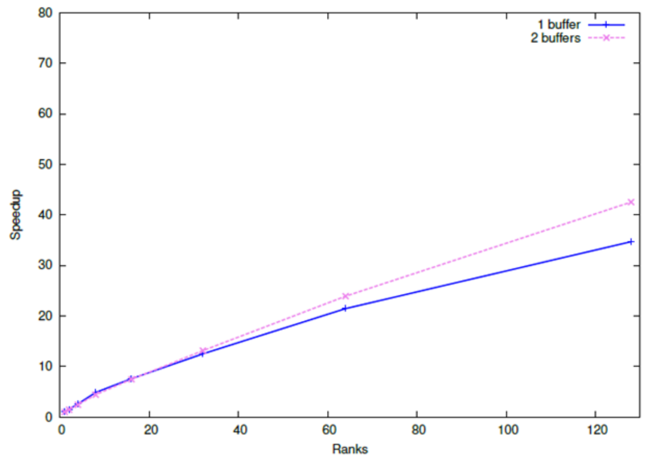

Fig. 6 Speedup of Graph500-MPI with One and Two Sending Buffers

## V. HYBRID EVALUATION

In this section, we present results from evaluating the hybrid OpenMP+MPI version of the Graph500 benchmark.

We will refer to this implementation as Graph500-HYB. For these results, we used both the PGOMP tool that we used for collecting data from the OpenMP version and the mpiP tool for MPI profiling. We did this separately so that the tools' overheads did not interfere with each other.

The Graph500-HYB code is quite a bit different than the Graph500-MPI version. Graph500-MPI is heavily centered on asynchronous MPI communication, but the Graph500-HYB code uses synchronizing all-to-all communication as it shares information between processes after a round of OpenMP parallel loop-based computation.

Fig. 7 shows that, not surprisingly, in simple raw performance (the internal Graph500 TEPS metric), Graph500-HYB is much better than the pure MPI version. Thus it has much less overhead and since most of Graph500-MPI's overhead was waiting for data, Graph500-HYB clearly removes much of the waiting time.

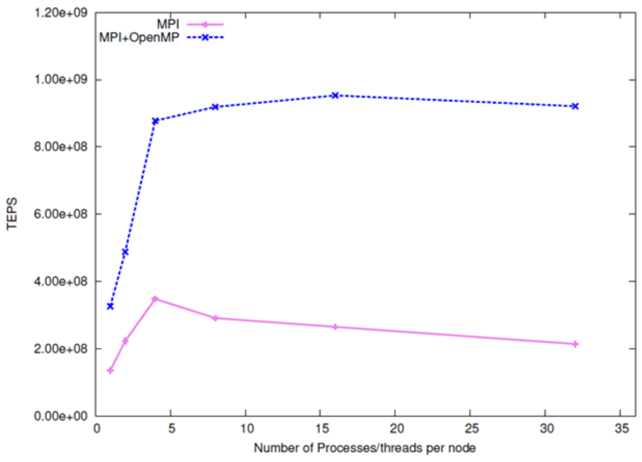

Fig. 7 Graph500-HYB and Graph500-MPI Performance on 8 Nodes

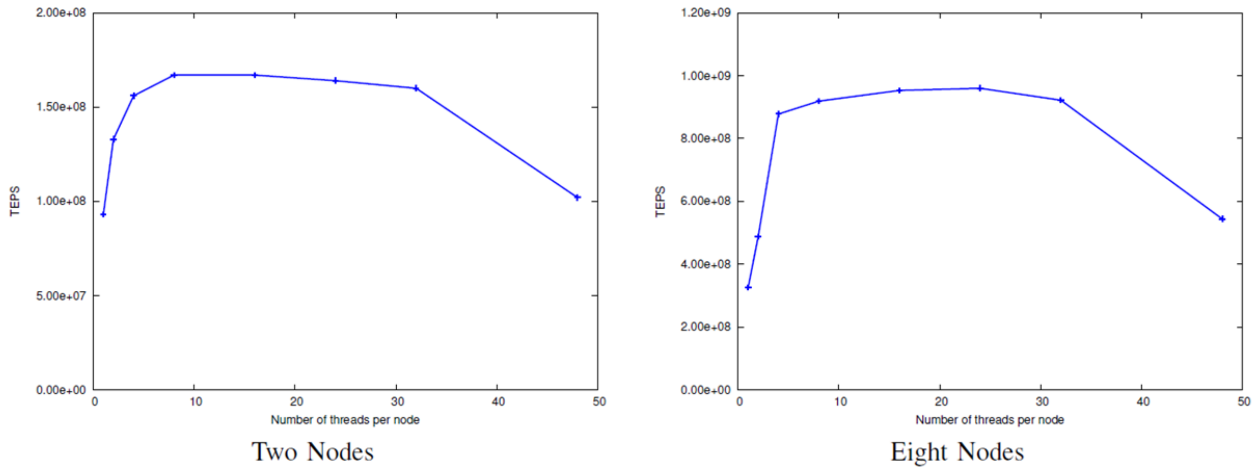

Fig. 8 Performance of Graph500-HYB on 2 and 8 Nodes, Varying Thread Count

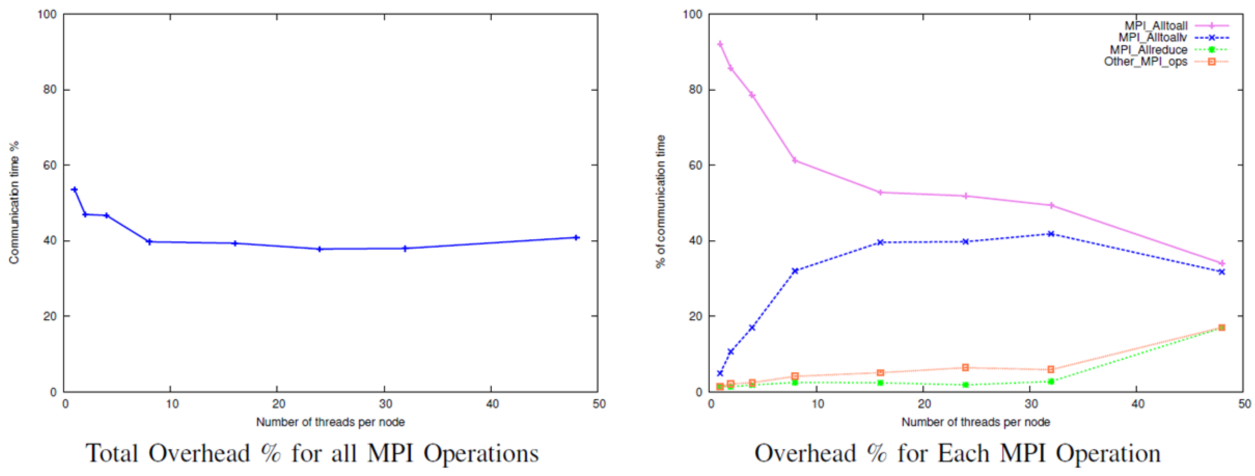

Fig. 9 Graph500-HYB MPI Overhead on 8 Nodes

Fig. 8 shows the performance of Graph500-HYB with varying thread counts per node, distributed over 2 nodes and 8 nodes. Both distributed configurations show the same general performance curve, with very few threads performing poorly, as expected, then a plateau up to 32 threads, with a tailing off of a performance at 48 threads. Recall that the platform these evaluations are performed on has 24 physical cores on each node, with 48 hardware hyperthreads. The surprising result here is that beyond 4-8 local threads, there is virtually no improvement in the performance of Graph500-HYB. Thus, its local parallelism is severely limited.

Fig. 9 shows the total MPI overhead and its breakdown into major components. Recall that Graph500-MPI consistently had about 70% of its time in MPI operations; Graph500-HYB shows a consistent 40%, thus reflecting its improved performance. Very low local thread counts have a skewed contribution between two all-to-all operations, and the balance between them slowly changes as thread counts continue to increase, but overall it is the two all-to-all operations that make up the majority of MPI overhead in Graph500-HYB.

Although Fig. 8 showed TEPS dropping off at 48 threads, this figure shows that MPI overhead remains consistent over thread counts, and thus the performance dropoff must be due to local issues. More importantly, since the MPI overhead remains consistent from about 8 threads to 48 threads, the lack of performance improvement across these thread counts signifies that all of the issues are local.

Recall that in the pure OpenMP version (Graph500-OMP), there was virtually no locking, and so the local overhead was mostly in barrier synchronization (end of a parallel loop) and the memory hierarchy (where implicit communication happens). With PGOMP we measured the barrier times, and although they do slightly rise as thread counts increase, we found them to be surprisingly small, accounting for less than 0.1% of the overall overhead. We did also use PGOMP to verify that other possible OpenMP overheads (library time, locking time) were negligible. We are therefore left to conclude that the local performance degradation is entirely due to the memory hierarchy.

Table IV summarizes the result of Graph500-HYB, putting together the observed MPI overhead measurements and the OpenMP barrier measurements, and then computing the not directly-measured memory hierarchy overhead and resultant productive computation percentage of application time, based on the observed TEPS performance for the different configurations. Recall that TEPS do not increase past about 8 threads per node, so productive computation drops drastically as thread count increases beyond this.

TABLE IV
OVERALL FACTOR CONTRIBUTION TO PERFORMANCE LOSS IN HYBRID GRAPH500

| Number of Threads | MPI Accounted Overhead % | Barrier/CS Accounted Overhead% | Memory System Overhead % | Computation % |
|---|---|---|---|---|
| 2 | 47.0 | 0.0 | 11.9 | 41.0 |
| 4 | 46.7 | 0.0 | 18.7 | 34.5 |
| 8 | 39.8 | 0.014 | 31.9 | 28.4 |
| 16 | 39.4 | 0.018 | 43.6 | 17.0 |
| 24 | 37.9 | 0.022 | 50.7 | 11.4 |
| 32 | 38.0 | 0.021 | 53.2 | 8.7 |
| 48 | 40.9 | 0.035 | 54.6 | 4.5 |

If our analysis holds, the end result is that as thread count increases, even with physical cores available (up to 24 on our platform), the memory hierarchy is basically thrashing (within cache and primary memory, not to disk), with threads causing coherency and caching conflicts at increasing rates. This seems reasonable since the MPI communication is performed by one single thread, which causes new data to be localized into its core's hierarchy, and then as all the threads access these data in a new parallel loop, the data not only must reach their core but also end up being irregularly shared as the threads traverse the local part of the graph. An analysis of this under memory simulation is needed to fully confirm this observation and deduction.

## VI. RELATED WORK

Roth et al. [6] performed somewhat similar analyses over particular Pthread-based parallel algorithms from PARSEC, categorizing the overhead factors into work, delay, and distribution. Most of their improvements were at the algorithm level, other than noting the potential effectiveness of an active barrier (spin barrier) for some applications over a passive (blocking) barrier.

Pavlovic et al. [7] investigated the memory system requirements that scientific applications have, concluding that future scientific computation needs radically new memory architectures if they expect to achieve good speedups. They used Gadget, MILC, WRF, and SOCORRO as their representative scientific applications, and focused on measuring memory bandwidth and cache effectiveness and the effect these have on the underlying CPI achieved by the workload.

Suzumura et al. [8] specifically focused on the performance characteristics of the Graph500 benchmark program in a distributed parallel environment; they optimized the implementation to achieve better speedup.

Harmony [9] is an LLVM-based tool that instruments Pthreads-based parallel programs. Their work noticed a related issue with the conventional Amdahl-based view of a parallel program, in that they found that code was shared between the sequential and parallel portions of the programs. The same group also investigates bottlenecks in a variety of application domains and aims to separate computation and memory system bottlenecks [10].

Murphy [11] performed an early study into the different behaviors that integer-based irregularly parallel algorithms have from the traditional matrix-based floating-point computations. Burtscher et al. [12] investigated the quantitative variation for irregular GPU-based programs. Other recent work has also quantitatively investigated performance issues in parallel algorithms, graph-based, and otherwise. Gill [22] investigated how the graph partitioning strategies, including Edge-Cuts, 2D block partitioning strategies, and general Vertex-Cuts, can reduce the required communication during the computation to synchronize node updates. They used the D-Galois system, a distributed-memory version of the Galois system based on the Gluon runtime, which implements partitioning-specific communication optimizations. Most of their improvements were at the application runtime. They concluded that the optimal partitioning strategy depends on the application, the input, and the number of hosts or scale. More prior work was done one parallel graph performance issues [13]-[15].

Various graph-processing frameworks and libraries have been and are being actively constructed, e.g. [16]-[18], [20],

[21], [23].

## VII. Conclusion

We presented an overview of studies into the factors that contribute to the overhead that causes parallel performance degradation in Graph500, taking it to be a representative application for graph-based problems. Detailed results can be found in the first author's thesis [3].

We measured the overhead contributed by traditional factors such as barrier synchronization costs, lock and critical section contention, and communication costs. In some settings, such as local parallelism, memory coherency costs could not be measured directly but were inferred from how much overhead was accounted for by other factors.

In the shared-memory parallelism of Graph500-OMP, variation in the sequential execution time was significant; ignoring it could lead to oversubscribing the overhead caused by the memory hierarchy. The Graph500-OMP version is limited in its scaling to one node (unless one considered cluster-wide virtualization techniques), so some element of distributed parallelism is needed.

For the pure MPI Graph500-MPI, which obviously does not take advantage of local node architecture, the graph search is bound by waiting for communication, and much of this seems inherent in the problem. Finally, the hybrid Graph500-HYB, which uses both MPI and OpenMP, performed far better than Graph500-MPI, but still consistently incurred about 40% of its time in distributed overhead costs (synchronized all-to-all communication). The most surprising aspect of Graph 500-HYB was how quickly its local parallel performance stopped improving. With nodes having 24 physical cores (48 hyperthreads) and 256GB of memory, it would seem that local parallelism would be extremely effective, but Graph500-HYB saw very little improvement above about 8 threads and saw some degradation with extreme thread counts (32-48).

In Graph500-HYB, barrier overhead was almost negligible and there is no locking overhead, so the overhead that reduces local parallelism effectiveness must be within the memory hierarchy. We did not measure, but did not see other explanatory causes such as threads sleeping. Its MPI overhead was consistent, but another problem organization that avoids all-to-all operations might be effective in reducing this. Nevertheless, local overhead begins to dominate at high thread counts.

For future work, we want to more deeply characterize performance issues in parallel graph algorithms, including using some real applications. Evaluating Graph500 under a memory simulator may be very costly but could provide real insight into exactly what is causing the largest performance impacts.

## Acknowledgment

We thank New Mexico State University for allowing us to use their machines. Also, we thank the reviewers for their time and helpful feedback.

## References

[1] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in Parallel Graph Processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[2] D. Bader, J. Berry, S. Kahan, R. Murphy, E. Riedy, and J. Willcock, "Graph 500 Benchmark 1 (Search)," 2010, www.graph500.org.

[3] M. El-Fituri, "Analyzing and Improving the Performance of Parallel Graph Algorithms, Ph.D. Thesis," New Mexico State University.

[4] M. Elfituri, J. Cook, and J. Cook, "Binary instrumentation support for measuring performance in openmp programs," in *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering*, ser. SE-CSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 19–23. (Online). Available: http://dl.acm.org/citation.cfm?id=2663370.2663374

[5] J. Vetter and M. McCracken, "Statistical Scalability Analysis of Communication Operations in Distributed Applications," in *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, 2001.

[6] M. Roth, M. J. Best, C. Mustard, and A. Fedorova, "Deconstructing the overhead in parallel applications," 2012 IEEE *International Symposium on Workload Characterization (IISWC)*, vol. 0, pp. 59–68, 2012.

[7] M. Pavlovic, Y. Etsion, and A. Ramirez, "On the memory system requirements of future scientific applications: Four case-studies," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, ser. IISWC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 159–170.

[8] T. Suzumura, K. Ueno, H. Sato, K. Fujisawa, and S. Matsuoka, "Performance characteristics of graph500 on large-scale distributed environment," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, ser. IISWC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 149–158.

[9] M. Kambadur, K. Tang, and M. A. Kim, "Harmony: Collection and analysis of parallel block vectors," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 452–463.

[10] M. A. Kim and S. A. Edwards, "Computation vs. memory systems: Pinning down accelerator bottlenecks," in *Proceedings of the 2010 International Conference on Computer Architecture*, ser. ISCA'10. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 86–98.

[11] R. Murphy, "On the effects of memory latency and bandwidth on supercomputer application performance," in *Proc. 2007 IEEE 10th Int'l Symp. on Workload Characterization*, ser. IISWC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 35–43.

[12] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," *2012 IEEE International Symposium on Workload Characterization (IISWC)*, vol. 0, pp. 141–151, 2012.

[13] R. Jongerius, P. Stanley-Marbell, and H. Corporaal, "Quantifying the common computational problems in contemporary applications," in Proc. *2011 IEEE Int'l Symp. on Workload Characterization, ser. IISWC* '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 74–.

[14] H. Inoue and T. Nakatani, "Performance of multi-process and multithread processing on multi-core smt processors," *in Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, ser. IISWC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[15] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proc. 2010 ACM/IEEE Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[16] E. Krepska, T. Kielmann, W. Fokkink, and H. Bal, "Hipg: Parallel processing of large-scale graphs*," SIGOPS Oper. Syst. Rev.*, vol. 45, no. 2, pp. 3–13, Jul. 2011.

[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146.

[18] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan, "Managing large graphs on multi-cores with graph awareness," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 4–4.

[19] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph

processing: Workload characterization on an ivy bridge server," in *Proc. of IEEE International Symposium on Workload Characterization* (IISWC), pp. 56--65, October 2015.

[20] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali, "Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory", *PVLDB 13*, 8 , 2020, 1304–13

[21] Laxman Dhulipala, Changwan Hong, and Julian Shun, ConnectIt, "A Framework for Static and Incremental Parallel Graph Connectivity Algorithms", *Proceedings of the VLDB Endowment*, 2021. To appear.

[22] G. Gill, R. Dathathri, L. Hoang, and K. Pingali. A Study of Partitioning Policies for Graph Analytics on Large-scale Distributed Platforms. volume 12 of PVLDB, 2018.

[23] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna, "A cache and memory-efficient framework for graph processing over partitions", In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming,* 2019, GPOP, (PPoPP'19). ACM, New York, NY, 393—394

[24] D. A. B. et al. Hpcs "scalable synthetic compact applications #2 graph analysis", version 2.2. http://www.graphanalysis.org/benchmark/HPCSSSCA2