Performance Analysis of a Large Memory Application on Multiple Architectures

Peter M. Kogge

University of Notre Dame, Notre Dame, IN, USA
kogge@cse.nd.edu

Abstract

The Graph500 Breadth-First Search benchmark has emerged as a well-documented PGAS-style application that both scales to large data set sizes and has documented implementations on multiple platforms over multiple years. This paper analyzes the reported performance and extracts insight into what are the leading performance limitations in such systems and how they scale with system size.

Contents

1 Introduction

The most common benchmark[2] to date for supercomputers has been the solution of dense linear equations of the form $Ax = B$, using the LINPACK package. The TOP500 web site[1] has been recording such measurements for almost 20 years, and ranked systems on the basis of their

[1]www.top500.org

reported sustained floating point operations per second, denoted R_{max}. Reports submitted for consideration in the listings usually provide not only R_{max} but also R_{peak} (the peak possible flop rate) and N_{max} (the dimension of the matrix at which the measurement was made).

A major objection to the generality of the TOP500 benchmark is that it is too regular, and too floating point intensive. A second objection is that it does not factor in data set size on a comparative basis. Detailed analyzes such as [5] indicate that in real, large, scientific codes the percentage of floating point is much less than in LINPACK, with address computations and memory referencing taking center stage. This trend is expected to grow even further as dynamic grids and multi-physics become more common.

In contrast, the Graph500 benchmarks[2] are meant to stress parts of a system not key to LINPACK, such as handling extremely large data structures that must encompass many physical nodes, and high amounts of unpredictable references into these structures. The current benchmark, **Breadth First Search** (**BFS**), involves creating a very large graph and then finding all vertices that are connected to some randomly chosen root vertex. The key performance metric is not flops but "Traversed Edges Per Second" (TEPS).

The only way to solve such problems, especially for the very largest graphs, is to employ a large parallel system where the major data structures must be partitioned over many nodes and embedded links are made between different vertices on arbitrary pairs of nodes, making this an archetypal PGAS-style problem.

After three years of measurements on now 100s' of systems, the key observation is that the peak system's TEPS improved by about 3,600X since the first listing, at a compound annual growth rate (CAGR) of 62X per year for the first 2.5 years, followed by what appears to be flattening. This is in contrast to the almost monotonous 1.9X CAGR observed in the TOP500 over 20 years.

The goal of this paper is to dive into this explosive growth rate and understand where the performance increases come from, and what class of systems are most effective in achieving them. In terms of organization, Section 2 defines the classes of architectures used in this study. Section 3 discusses the Graph500 benchmark. Section 4 discuss the range of typical implementations of the benchmark on such systems. Section 5 overviews the as-reported Graph500 results in terms of architectures that support them. Section 6 discusses how these results reflect on scalability in the underlying systems. Section 7 looks in detail on a series of implementations using two lightweight systems. Section 8 concludes.

2 Architecture Classes

In analyzing the future of supercomputers, the Exascale report[4] defined two general classes of architectures that have shown up in TOP500 rankings, **heavyweight** and **lightweight**. Since then the **hybrid** class has appeared in the TOP500. Fig. 1 pictures typical nodes for these classes.

All three classes have also appeared in the Graph500 rankings, with the addition of several more specialized, but more PGAS-relevant architectures. Each class is discussed briefly below.

In these descriptions, a "socket" refers to a high density logic chip such as a microprocessor that performs some major function, a "core" refers to a logic block capable of independently executing a program thread, and a "node" is that collection of sockets, memory, and other support logic that make up the minimum-sized replicable unit in a parallel system.

[2]www.graph500.org.

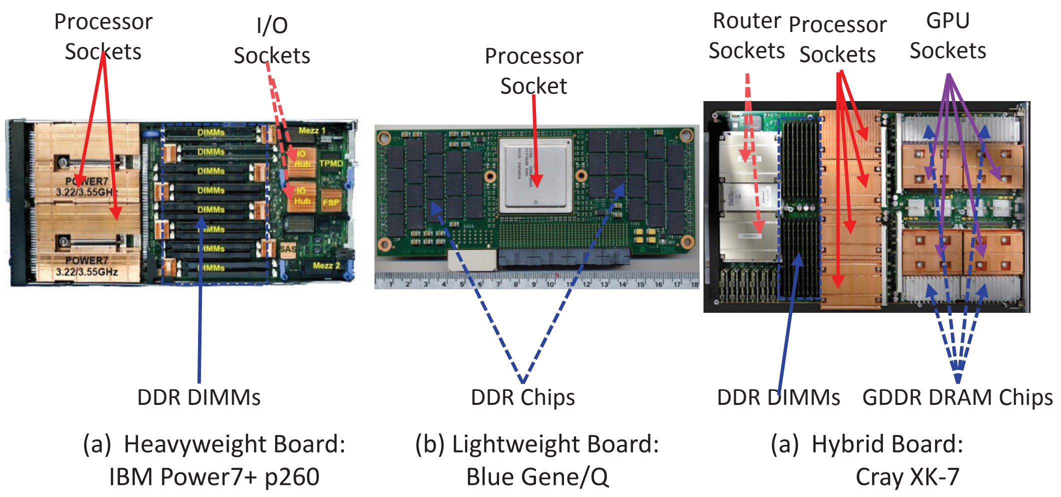

Figure 1: Typical Node Boards for different Architecture Classes.

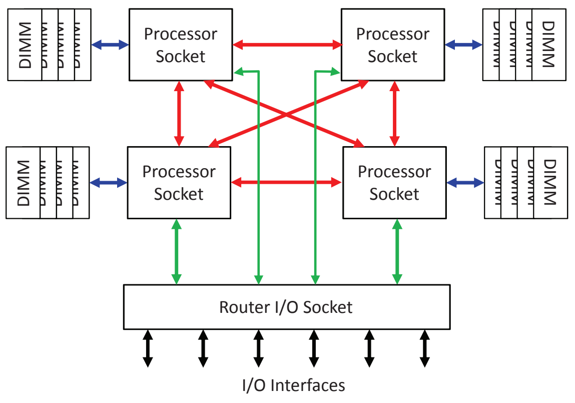

Figure 2: Modern Heavyweight Node Architecture.

2.1 Heavyweight Architectures

Heavyweight architectures are the natural progression of what are now the ubiquitous multi-core microprocessors, and are designed to work with a combination of support chips to provide the most possible performance per chip, typically at high clock rates and with a fairly large amount of memory. Fig. 2 diagrams a typical heavyweight node, with Fig. 1(a)[3] a sample node from a POWER7+ system, where much of the board is taken up by the heatsinks for the processor sockets and the I/O sockets.

The chips in modern heavyweight processor sockets have 8-16 cores, each with 2-4 FPUs, and run at or above 3GHz. Typically up to 4 independent memory channels are supported, with each channel supporting high capacity, multi-rank, high bandwidth DIMMs.

All the cores in a single heavyweight processor socket share access to all the memory attached to that socket, usually including cache coherence. In many heavyweight designs this sharing

[3]image from http://www.theregister.co.uk/2012/11/13/ibm_ power7_plus_flex_storwize/.

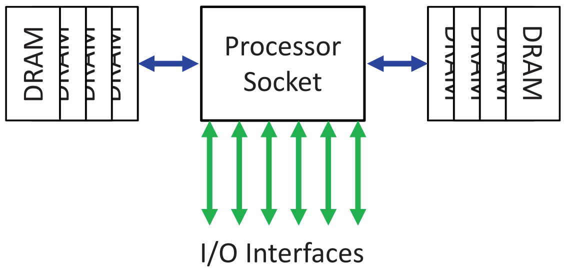

Figure 3: Modern Lightweight Node Architecture.

Parameter	L	P	Q	Q/P
Cores/node	2	4	16	4X
Core Clock (GHz)	0.7	0.85	1.6	1.9X
Max Node Memory (GB)	1	4	16	4X
Memory Ports per Node	1	2	2	same
Memory B/W per Port	5.6	6.8	21.35	3.1X
Total Memory B/W (GB/s)	5.6	13.6	42.7	3.1X
Inter-node Topology	3D	3D	5D	
Links per Node	12	12	22	4.7X
Bandwidth per Link (GB/s)	0.175	0.425	2	4.7X
Total Link B/W (GB/s)	2.1	5.1	44	8.6X

Table 1: BlueGene Family Characteristics.

of memory extends to some, usually small, number of other compute sockets in the same node. These compute sockets typically share an intra-node network. Thus a single node has a moderately large number of cores that share local node memory with each other, but has a more distributed memory interface when dealing with any core or memory elsewhere in the system, requiring software such as MPI for communication.

Typically at most a 100 or so such nodes can fit in a single rack.

2.2 Lightweight Architectures

The introduction of the IBM Blue Gene/L[6] in 2004 used a compute socket with a dual core processor chip that included a memory controller, I/O, and routing functions on a single chip. The cores were much simpler than for the heavyweight machines, and ran at a much lower clock rate. Such a chip, when combined with some memory, made a complete node in the above sense, as pictured in Fig. 3. Because the required heatsink was so much smaller than for a heavyweight, many of these small cards could be packaged in the same space as a heavyweight node (up to 1024 of such nodes in a single rack). Subsequent versions, Blue Gene/P[1] and now Blue Gene/Q[3], as pictured in Fig. 1(b)[4], have continued this class of architecture. Blue Gene/Q in particular has a 16+1 multi-core processor chip, with two memory controllers connected directly to conventional DDR3 DRAM chips on the same node board. Table 1 summarizes some of the key parameters of these systems.

[4]image from http://www.cpushack.com/wp-content/ uploads/2013/02/IBM51Y7638_BlueGeneQ.jpg

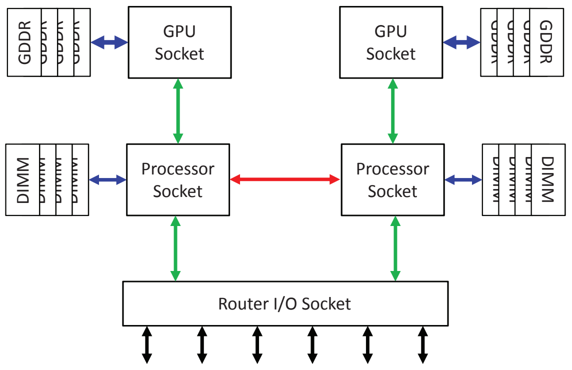

Figure 4: Modern Hybrid Node Architecture.

Again all cores in a node's processor socket view the node as a single shared memory structure, but as with the heavyweight nodes, other nodes are viewed in a distributed fashion, requiring software such as MPI for communication.

2.3 Hybrid Architectures

The original Exascale report[4] identified only the heavy and lightweight classes. Since then, a third class has surfaced in the Top500, which combines with a heavyweight socket a second compute socket where the chip in it boasts a large number of simpler cores, usually with an even larger number of FPUs per core. Today such chips are derived from **Graphics Processing Unit** (**GPUs**), such as the Nvidia Tesla architecture[5], the Intel Xeon Phi architecture[6], or the AMD GCN architecture[7].

Fig. 1(c) pictures one such node from the Titan supercomputer[8].

As with the heavyweight nodes, something of on the order of a 100 such nodes could be packed into a single rack.

Memory within such nodes is today usually not as fully shared as in the other classes. Instead, the accelerator node typically can only access its own local GDDR (Graphics Double Data Rate) memory during computation. Thus the heavyweight host processor must explicitly transfer between the accelerator's memory and its own memory. This staging takes both time and program complexity, and derates the usefulness of the accelerator when random accesses to larger amounts of data than can fit in the GDDR is needed.

2.4 Other Architecture Classes

In addition to systems that fall into the above three classes, two other system families have shown up in the Graph500 rankings. First is the Cray XMT-2 massively multi-threaded system. This architecture is particularly relevant to the GRAPH500 problem because it natively supports a large PGAS shared memory model, when a core anywhere in the system can directly access a memory anywhere else, without intervening software.

[5]http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries- Overview-LR.pdf
[6]http://www.intel.com/content/www/us/en/processors/xeon/ xeon-phi-detail.html
[7]http://www.amd.com/us/Documents/GCN_Architecture_ whitepaper.pdf
[8]http://techreport.com/r.x/ 2012_10_29_Nvidia_Kepler_powers_ Oak_Ridges_supercomputing_Titan/titan-blade.jpg

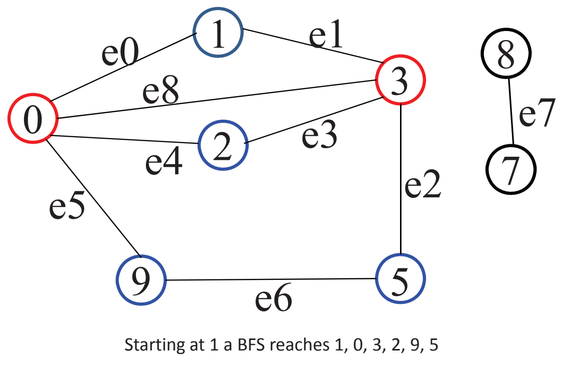

Figure 5: A Sample Graph.

The second system architecture consist of variations of the Convey FPGA-based systems[9]. This architecture is also particularly relevant because its memory system, albeit smaller than that possible with large clusters, has much more internal bandwidth, making it a good match again to BFS.

3 The BFS Benchmark

Several benchmarks are planned under this Graph500 umbrella, with only one of them, Breadth First Search (**BFS**), currently defined and tracked through several generations of systems. Two other benchmarks (Shortest Path and Maximal Independent Set) are planned in the near future. Unlike the scientific-orientation of LINPACK, these benchmarks are believed to be highly related to areas such as cybersecurity, medical informatics, data enrichment, social networks, and symbolic networks.

The purpose of the kernels defined in BFS is to build a very large graph, and then start at any random vertex and identify all other vertices that are connected to it.

There are three major steps in the benchmarking process:

1. Graph construction: create a data structure to be used for the BFS. The two major configuration parameters that go into this are:
   - Scale: base 2 log of number of vertices (N) in the graph.
   - Edgefactor: ratio of total number of edges to total number of vertices in the graph.
2. Breadth-First Search: starting at a random vertex, follow all edges from that vertex to all vertices reachable over a single edge, and repeat from each vertex that had not been touched before until as many vertices as possible have been reached.
3. Validation: check that the answer is correct.

Fig. 5 shows a sample graph. Starting at vertex 0, a valid BFS output would be 0, 1, 2, 9, 3, 5. Starting at 2 a valid output would be 2, 3, 0, 9, 5, 1.

If M is the total number of edges within the component traversed by a BFS search (the second step), and T is the time for doing that search, then the key reported Graph500 metric

[9]http://www.conveycomputer.com/

Category	Level	Scale	Vertices (Billion)	Memory (TB)
Toy	10	26	0.1	0.02
Mini	11	29	0.5	0.14
Small	12	32	4.3	1.1
Medium	13	36	69	17.6
Large	14	39	550	141
Huge	15	42	4,398	1,126

Table 2: GRAPH500 Problem Size Categories.

is **Traversed Edges per Second** (**TEPS**), computed as is M/T. The time for the first and third steps is not part of the benchmark.

As with the TOP500's N_{max}, there is a problem size component to the GRAPH500, although in this case it is far more important to evaluate the success of a system than N_{max} is to TOP500. Table 2 lists different classes of problem sizes and a typical memory footprint of the resulting data structure in bytes, assuming an average node and its associated edges (on average 32 of them) take about 282 bytes. The "Level" in this table is approximately the base 10 log of the estimated size in bytes, and "Scale" is the base 2 log of the number of vertices. To date, very few systems have reached the "Large" category (needing 140TB), let alone the "Huge" category.

Note in Table 2 the scale in each problem category goes up by 3 (a growth in size of 8X) as the level goes up by 1 (a growth by a factor of 10), except between levels 12 and 13, where there is a growth by 16X in size. This extra step is to account for the difference between 8^3 and 10^3.

4 BFS Implementations

The Graph500 web-site gives reference codes for both the graph generators and the BFS algorithm itself. For the latter, several different reference codes are given, some of which are described in the subsections below.

4.1 Sequential Code

The reference sequential code (in file seq_csr.c) has several very large data structures, where N is the number of vertices ($N = 2^{scale}$, which for toy problems is 64 million and for huge problems is 4 trillion) and M the number of edges in the graph (for Graph500, $M = 32N$):

- An array *xoff* of size 2N has two words per vertex, indexing to the start and end of a list in *xadj* that includes the edges leaving that vertex.
- An array *xadj* of length M, where each word holds the destination vertex of an edge.
- An array *bfs_tree* of length N, where the v'th entry corresponds to vertex v, and gives either the index to v's " parent" vertex in terms of the BFS search or a special code if v has not yet been touched.
- An array *vlist* of length N that contains the vertices in the order in which they were touched during BFS, with vertices that were touched in the same "level" of the search arranged in sequential order.

The sequential code is a triply nested loop where:

- the innermost loop iterates over all edges exiting a vertex touched in the last level, determining if the vertices on the other side have been touched before, and if not marking them as touched in *bfs_tree* and adding them to *vlist* as part of the next level,
- the middle loop iterates over all vertices touched in the last level of the BFS search,
- the outer loop repeats the inner two as long as new vertices were added to a new level.

Overall time complexity is on the order of O(N+M).

4.2 XMT Code

The XMT reference code demonstrates the advantages of programming a PGAS problem in an architecture where all memory is visible to all threads. It is virtually identical to the sequential code with two exceptions: iterations of the inner two loops of the BFS are performed in parallel by different XMT threads, and the updates to *bfs_tree* and *vlist* are done with atomic operations that guarantee that if two threads try to touch the same vertex at the same time, only one wins.

4.3 MPI Codes

The BFS problem quickly scales to problem sizes that will not fit in the memory of any single node of any architecture class. While the XMT does scale to bigger sizes with multiple nodes, its current implementations still don't grow large enough in total capacity to match the higher scales. Thus a third reference implementation partitions the data structures onto different cluster nodes and uses explicit MPI calls to create the equivalent of a PGAS implementation.

The approach taken in the reference codes is to partition the equivalent of *xoff* and *vlist* into approximately equal-sized pieces, one per node. *xadj* is also partitioned so that all edges for all vertices on a node are also on the same node. When the BFS code running on a node encounters an edge to a vertex that needs to be touched, the code explicitly computes on which node the vertex resides, and dispatches MPI messages to so inform the other node.

Besides the explicit computation of which node hosts a vertex, this code also handles differently the process of identifying which vertices have been recently touched, and thus are to be explored in the next level. A bit vector is kept on each node with one bit for some number of local vertices. When any node follows an edge to a vertex, the associated bit on the appropriate node is set to 1 atomically via an MPI OR sent to that node.

Two copies of these bit vectors are kept, one to record newly touched vertices and one from the last level to indicates which vertices were touched on the last level. At the end of each level, these vectors are reversed, and the one holding the previous (and now explored) markings reset to all zeros. This is to avoid the complexity of remotely enqueuing a newly-touched vertex on a remote queue.

In addition, the two states of a vertex of being touched (and having a parent), and untouched are expanded to three states: untouched, touched and explored, and touched but not yet explored. An MPI atomic MIN operation sent to the target node does both a check and state transition.

There are thus two MPI atomic operations per edge traversed: set the bit and do a MIN. Transit times and overheads for these messages greatly outweigh the execution costs at their destination, making them the primary gate on TEPS.

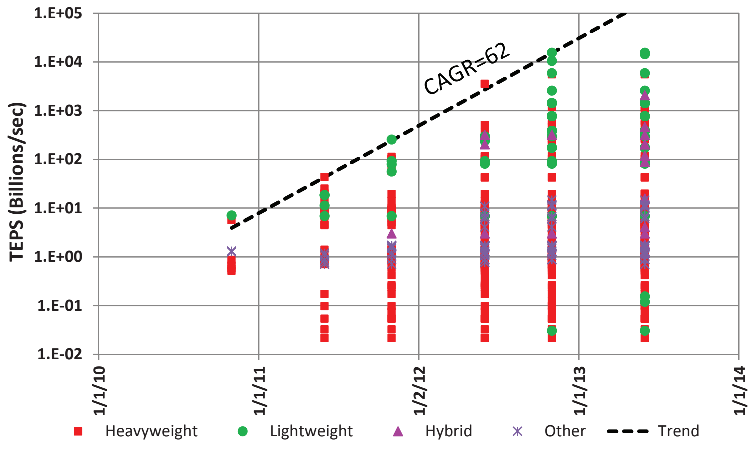

Figure 6: TEPS over time.

Finally, gluing this together is a series of barriers to synchronize all nodes between levels, and to determine when the last level has been reached and there are no more vertices to explore.

As will be seen in Section 7, over time there has been a significant increase in performance due to algorithms only. The most common approach is to sort updates to other nodes by node number, and send single packets with multiple vertex updates attached, thus amortizing overhead. Packing vertices by path may also be used so that updates that are "on the way" to some remote node can piggyback and not consume independent bandwidth.

5 Graph500 Results vs Architecture

A ranking for a system in the Graph500 provides a description of the system used and a subset of key performance parameters. The former include primarily core and node count, total memory, and a verbal description of the cores and sockets. The latter include the TEPS rate, the scale of the problem solved, and a classification of the algorithm used. For this paper, all such listings were downloaded and augmented where possible with additional information about each system, with the primary addition being a tag indicating the class of architecture into which the system falls. This tag was used to select a symbol to use on all the scatter plots that follow:

- Red squares are heavyweight systems.
- Green circles are lightweight systems.
- Purple triangles are hybrids.
- Brown stars are other.

Fig. 6 graphs the TEPS rate over time for all reported systems. The key observation is that the peak systems' TEPS have improved by about 3,600X from the first listing in Nov. 2011 and Nov. of 2012, at a nearly constant compound annual growth rate (CAGR) of 62X per year, but this appears to have flattened as of June 2013. It is also interesting to see that until Nov. 2012 there was a noticeable mix of architecture classes near the top of the rankings, but after this, all points in the top order of magnitude are lightweight.

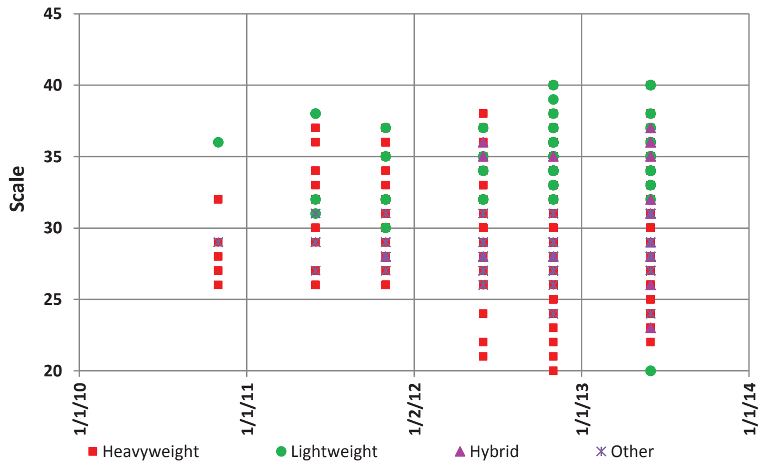

Figure 7: Scale over Time.

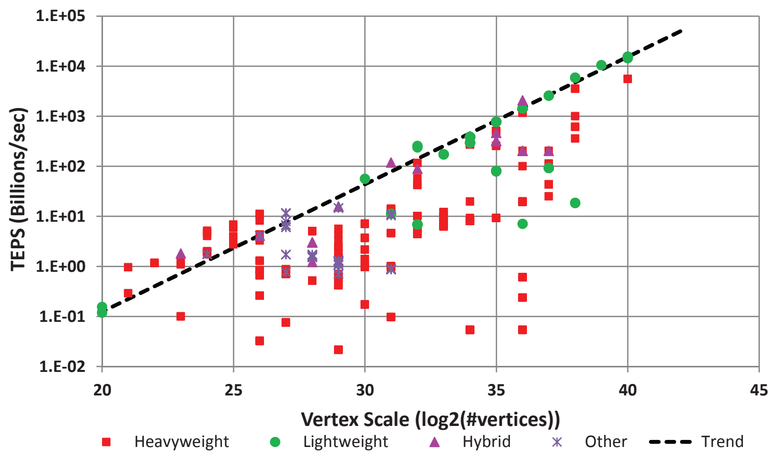

Figure 8: TEPS versus Scale.

A second observation is the now six order of magnitude range of performance in the most recent rankings, in comparison to the only 230X difference between number 1 and number 500 for the Top500. The reason is that there is for the Graph500 a lot more interest in performance *as a function of scale*. Fig. 7 provides a time-denoted diagram of scale to match Fig. 6. While it has nowhere near the CAGR, again nearly all of the biggest problems solved by size are solved on lightweight systems.

This dominance of lightweight systems is seen again clearly in Fig. 8. After a scale of 30 (1 billion vertices), lightweight systems are uniformly at the peak of the performance metric at each scale.

Also interesting in Fig. 8 is that the dotted line in this graph is proportional to 1.8 to the power of scale. Given that the number of vertices is 2 to the scale, this means that performance grows slightly more slowly than the size of the problem solved. This will be discussed more in the next section.

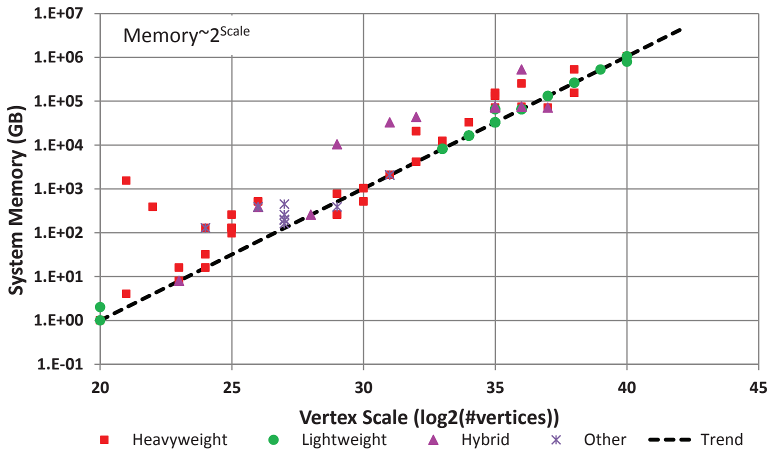

Figure 9: Memory Capacity versus Scale.

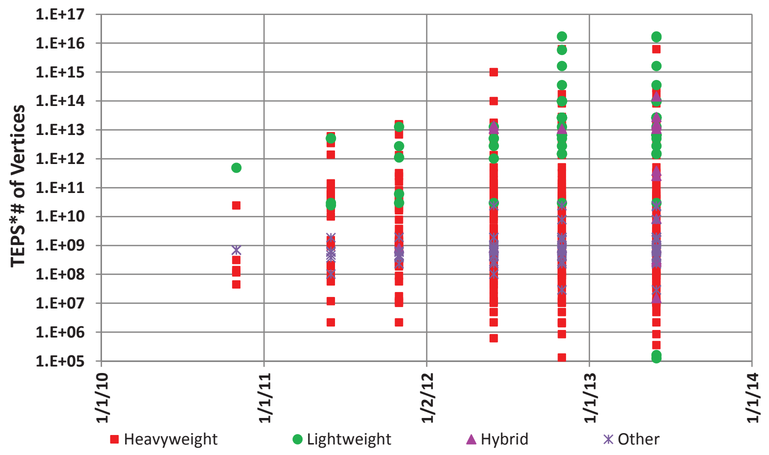

Figure 10: TEPS*Size over Time.

Fig. 9 then graphs the memory capacity of the reported systems versus the maximum size problem they solved. The close clustering of the points to the dotted line indicates that most systems did in fact try to select problem sizes that came close to filling memory.

Finally, to couple the two key metrics, TEPS and scale, Fig. 10 graphs the product of the TEPS rate and the number of vertices versus time. Peak values here are growing faster than linearly with time, indicating that both metrics are improving in concert.

6 Scalability in the Graph500

The key questions to be addressed in this section are "what drives the joint growth in both TEPS and scale," and "how does this vary among architecture classes."

The most obvious place to look is in the inherent parallelism in the system, starting with

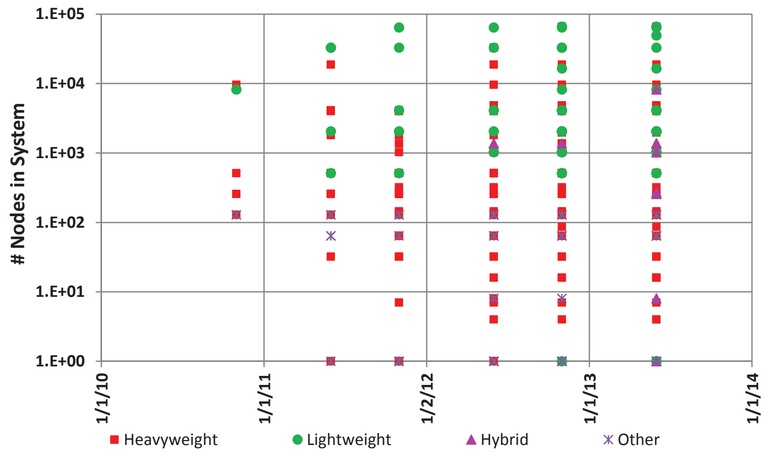

Figure 11: Node Count Growth in the Rankings.

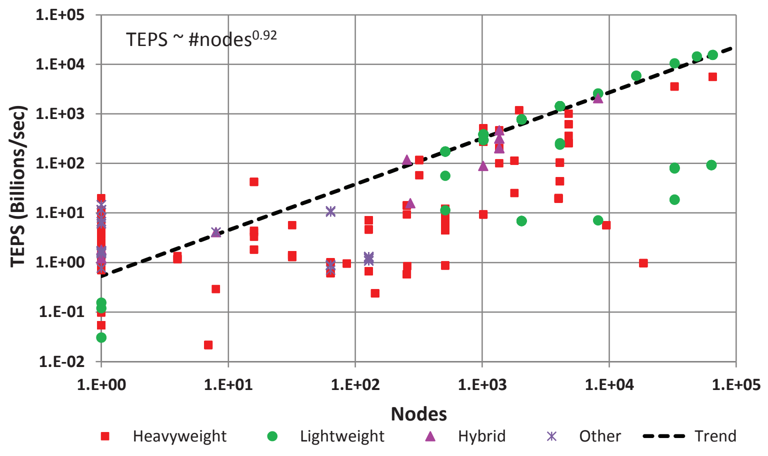

Figure 12: TEPS as a Function of Node Count.

the number of nodes. Fig. 11 plots the growth in node count through the rankings, where we see an order of magnitude growth in peak node count over the first year, and then a flattening.

6.1 Weak Scaling

Fig. 12 graphs TEPS as a function of just this node count, ignoring all other factors such as core count, clock rates, or time of listing.

The most interesting observation is that the best of breed has a TEPS rate proportional to the number of nodes to the 0.92 power. Further, most of these are lightweight systems, with heavyweight systems often delivering 10 to 100 times less performance at the same node count. Further, most of the leading lightweight points are BlueGene/P systems, designs with very significant amounts of inter-node bandwidth. The obvious conclusion is that, with sufficient bandwidth, BFS has almost perfect weak scaling, with the slight nonlinearity (exponent of 0.92 rather than 1) due to such things as barriers or sync points (again something for which the

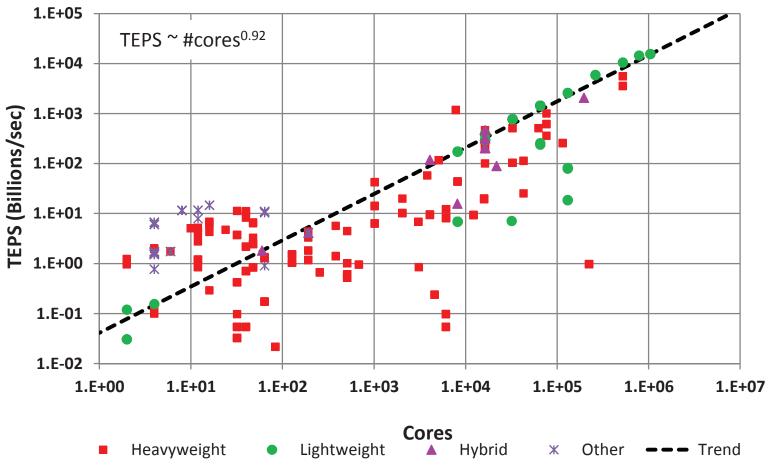

Figure 13: TEPS as a Function of Core Count.

BlueGene has significant hardware support).

The other observation from Fig. 12 is the significant performance from single node systems. It isn't until we get in excess of 32 nodes that parallel systems outdo the best of the single nodes. This is again most probably related to bandwidth, with memory bandwidth now replacing inter-node bandwidth. This is particularly easy to see in the best of the single nodes, which are Convey systems that have a very high bandwidth partitioned memory system within them. It will be interesting to see how such systems scale when more nodes are employed, and the bandwidth between nodes becomes more conventional.

Fig. 13 is a similar graph except that it plots cores rather than nodes. This distinction expands what was the "1 node" peak before, and we see a region of single node, multi-socket, multi-core points where the system is a single shared memory system and we are replacing node-node bandwidth with multi-bank memory bandwidth.

6.2 Incremental Performance

A related way to look at the performance results is to look at the incremental growth in performance offered by each node as we grow system sizes. Fig. 14 divides the total TEPS rate of each listing by the number of nodes, and plots this versus time. This new metric provides insight into the incremental power of each additional node.

The key observation here is the over 1000X increase we see on a per node basis over the first 1.5 years, followed by an essential flattening over the last 18 months.. While some of this improvement is due to improved hardware, a more likely explanation is that the algorithms used have improved significantly (this is discussed in more detail in the next section).

The other observation is about how different architecture classes compare. In almost each ranking, the highest per node performance has been by a single node Convey box, with its large internal bandwidth. Also, the lightweight class, which provides the highest overall performance, are middle of the pack in terms of per node performance. This is amplified by Fig. 15, which expresses the per node performance in terms of node count. Here it is obvious that the lightweight architecture becomes much more scalable as we move to larger systems.

Note also in this figure that the lightweight systems from about 1,000 to 100,000 nodes have

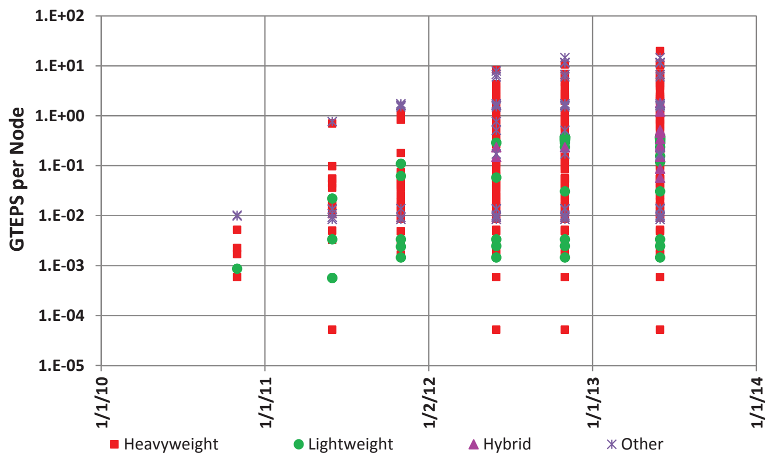

Figure 14: TEPS per Node over Time.

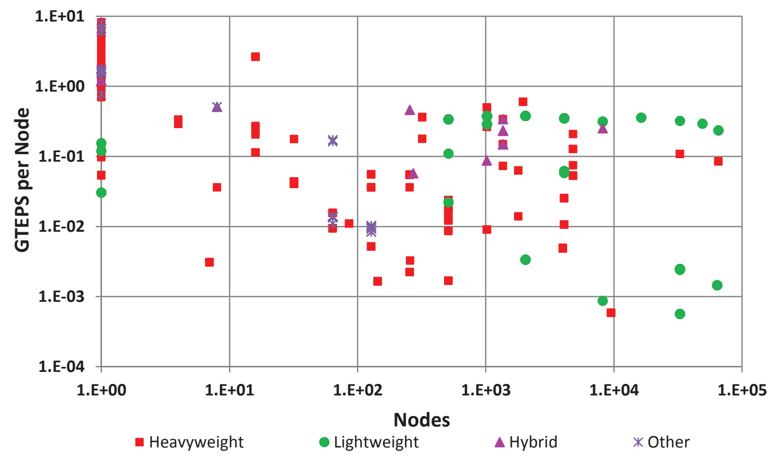

Figure 15: TEPS per Node versus Node Count.

a slight decrease in TEPS per node as node count increases. This is further evidence of the effects of serializing and decreased injection bandwidth because of cross-node traffic.

7 Detailed Analysis of Lightweight Implementations

Given the high performance of lightweight systems in these rankings, and that there are multiple systems with potentially different codes and different numbers of nodes but the same node design, a great deal can be learned by isolating on just them. In particular, we partition the data points into BlueGene/P and /Q systems. Figs. 16 and 17 look at TEPS per node as a function of size and time respectively.

Fig. 17 shows that after the first ranking, the best TEPS per node for BlueGene/P was fairly constant, whereas there was about a 17X gain for BlueGene/Q before it too stabilized. Given

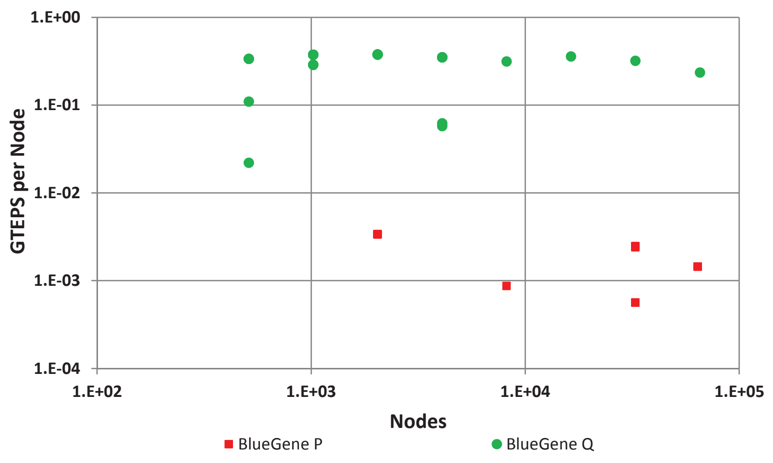

Figure 16: BlueGene TEPS per Node versus Node Count.

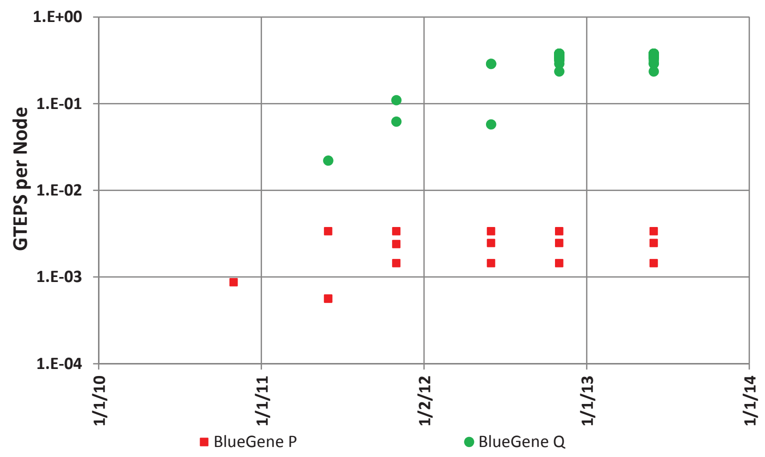

Figure 17: BlueGene TEPS over Time.

that all the Q points have identical hardware, this increase can only be explained as optimizing the algorithms. It will be interesting to see if there are further gains in later rankings.

It is also interesting to note that there is a 112X difference between a /Q and a /P once both have reached a stable value. Earlier, Table 1 listed the hardware differences between P and Q, along with the change from P to Q. The biggest difference is in aggregate off-node bandwidth. Q has 4.7 times the number of links, with 8.6 times bandwidth per link, for a total of 40 times the total bandwidth. The difference in memory bandwidth is about a factor of 3.1. The rest must be due to better algorithms.

8 Conclusions

This paper has explored the results of several years of listings of performance of the BFS algorithm on a wide variety of architecture classes, where performance has two components: TEPS and graph size. TEPS performance does seem to exhibit significant performance improvements possible by algorithmic changes to optimize the use of bandwidth. Problems of small size are best processed on single-node systems with a lot of shared memory and a lot of memory bandwidth, such as found in the Convey systems. Problems of large size are best run on lightweight systems where a large number of nodes can be used. Unlike TOP500, hybrid systems are nowhere near the top in terms of performance.

In terms of details, the best systems seem to follow a near perfect weak scaling model, with an exponent of almost 1 (0.92) relating the number of nodes to performance.

Looking ahead, it will be interesting to see if performance for existing systems such as BlueGene/Q continues to improve in ways that indicate improvement in basic algorithms. Also, with the imminent release of a "Green Grap500" listing, an energy analysis akin to what has been done for LINPACK can begin. Also, it will be interesting to see if the results for BFS carry over for both the upcoming additional Graph500 benchmarks and other PGAS benchmarks. Finally, it will be interesting to see how inherently PGAS architectures such as XMT may evolve, and if they can match or exceed the performance of other architecture classes.

8.1 Acknowledgements

This work was supported in part by the University of Notre Dame and in part by Sandia National Laboratory, Albuquerque, NM under both a contract with the U.S. DoD DARPA as part of the UHPC program, and a DOE-funded extreme computing effort called XGC.

References

[1] IBM Blue Gene team. Overview of the IBM Blue Gene/P project, 2008.

[2] Jack J. Dongarra. Performance of various computers using standard linear equations software. *SIGARCH Comput. Archit. News*, 20(3):22–44, June 1992.

[3] R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, G.L.-T. Chiu, P.A. Boyle, N.H. Chist, and Changhoan Kim. The IBM Blue Gene/Q compute chip. *Micro, IEEE*, 32(2):48 –60, march-april 2012.

[4] Peter M. Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems. Technical Report CSE 2008-13, Univ. of Notre Dame, Sept. 2008.

[5] R.C. Murphy and P.M. Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *Computers, IEEE Transactions on*, 56(7):937–945, 2007.

[6] The BlueGene/L Team, T Domany, Mb Dombrowa, W Donath, M Eleftheriou, C Erway, J Esch, J Gagliano, A Gara, R Garg, R Germain, Me Giampapa, B Gopalsamy, J Gunnels, B Rubin, A Ruehli, S Rus, Rk Sahoo, A Sanomiya, E Schenfeld, M Sharma, S Singh, P Song, V Srinivasan, Bd Steinmacher-burow, K Strauss, C Surovic, Tjc Ward, J Marcella, A Muff, A Okomo, M Rouse, A Schram, M Tubbs, G Ulsh, C Wait, J Wittrup, M Bae, K Dockser, and L Kissel. An overview of the bluegene/l supercomputer, 2002.