Many-Core Graph Workload Analysis

Stijn Eyerman
Intel Corporation
Kontich, Belgium
stijn.eyerman@intel.com

Wim Heirman
Intel Corporation
Kontich, Belgium
wim.heirman@intel.com

Kristof Du Bois
Intel Corporation
Kontich, Belgium
kristof.du.bois@intel.com

Joshua B. Fryman
Intel Corporation
Hillsboro, OR, USA
joshua.b.fryman@intel.com

Ibrahim Hur
Intel Corporation
Hillsboro, OR, USA
ibrahim.hur@intel.com

***Abstract*—Graph applications have specific characteristics that are not common in other application domains and therefore require thorough analysis to guide future graph processing hardware design. In this paper, we analyze multiple graph applications on current multi and many-core processors, and provide conclusions and recommendations for future designs. We restate well-known characteristics of graph applications, such as a low compute to memory ratio and irregular memory access patterns, but we also provide new important insights on executing graph applications on many-core processors.**

Our main novel observations are (i) some memory streams do show locality, while others show no locality, (ii) thread imbalance becomes a major problem with many threads, and (iii) many threads are required to saturate high-bandwidth memories. The first observation calls for a selective memory access policy, where accesses with locality are cached and prefetched, while accesses without locality can remain uncached to save cache capacity, and can fetch only one element from memory instead of a full cache line to save on memory bandwidth. The last two observations are contradicting: more threads are needed, but they are not used efficiently due to thread imbalance. Our recommendation is therefore to thoroughly revise the graph analysis algorithms to provide more scalable parallelism to be able to exploit the potential of many-core architectures with high-bandwidth memory. In addition, providing a few high-performance cores can speed up sections with low parallelism.

***Index Terms*—graph applications, workload analysis, many-core processors**

I. Introduction

Graph applications for analyzing big data sets are gaining importance. Many data sets can be represented as graphs, where each vertex corresponds to an object and each edge represents a relation between two objects. Examples are social networks, road networks, physics models and co-preference graphs. These graphs can be very large with millions to billions of vertices and a multiple thereof of edges. Furthermore, the reality modeled by these graphs changes quickly, so the graphs need to be constantly updated, after which the analysis has to run again, ideally presenting results (e.g., recommendations) in real time. Therefore, researchers and industry are looking at developing high performing graph analysis software and hardware [1]–[5].

Graph applications are different from the conventional high-performance scientific applications in that they are highly irregular [6]. This is because graphs are inherently sparse: the number of edges is a small fraction of the total amount of possible connections. Therefore, the edges are not represented as a sparse adjacency matrix, but instead as a list of neighbors for each vertex. This list of neighbors has no regularity, causing scattered accesses all over the graph data structure. This behavior ruins memory locality and the predictability of memory accesses and branch outcomes. Therefore, caches, prefetchers and branch predictors have a low hit rate, eliminating the main performance boosting techniques of today's high-performance processors. Designing processors that are optimized for high-performance graph analysis is therefore a challenging task.

In this paper, we analyze the performance of graph analysis kernels on recent multi and many-core hardware (Intel Xeon and Xeon Phi), and project how a potential future many-core architecture would perform. We analyze the main performance bottlenecks, and make some recommendations on how future graph analysis targeted processors can optimize performance for these applications.

Our findings confirm existing insights, such as irregular accesses, and poor cache, prefetcher and branch predictor efficiency, but we also gain new insights as these applications enter the many-core era. Our main novel findings are:

- Some of the memory accesses do show locality, and benefit from caching and prefetching. A mechanism to handle accesses with and without locality differently has the potential to largely improve cache efficiency, while keeping its performance benefits.
- Because of the high cache miss rate, most applications are memory bound. Providing high-bandwidth (on-chip) memory can therefore boost performance, but in order to saturate this bandwidth, access latency has to be kept low, and a sufficient amount of threads need to be active to generate many concurrent accesses.
- At high thread counts, algorithms that scale well on current hardware start to scale worse because of load imbalance and limited parallel tasks. Algorithms should be revised to exploit more parallelism. Additionally, a few high-performance cores can accelerate phases of the application with low parallelism.

II. Related Work

A. Graph Application Frameworks and Benchmark Suites

Several graph frameworks have been developed that relieve the programmer from graph representation and data distribution details, in order to increase productivity. Graphlab [7]

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

is a vertex-centric framework, where graph algorithms must be expressed as a program running on a vertex, and on its edges and neighbors. GraphBlas [3] uses sparse matrix operations to perform graph algorithms. SociaLite [8] is a database oriented graph framework, where operations are specified in a declarative language. Galois [4] is a framework for parallelizing irregular applications, and is therefore ideal for graph applications. Giraph [9] runs on top of Hadoop and is message based, while GraphX [5] is part of Apache Spark, running in-memory distributed graph applications. Satish et al. [10] compare multiple frameworks, and conclude that Galois performs closest to a native implementation in C++, which performs best.

The lack of a standard benchmark suite for graph applications has inspired multiple concurrent benchmark suite proposals: GAP [11], GraphBIG [12] and CRONO [13] have all been proposed in 2015. Pollard and Norris [14] have compared multiple benchmarks and frameworks, and conclude that on average, the GAP benchmark suite performs best on a multicore processor. Therefore, we use the GAP Benchmark Suite [11] for our analysis. GAP consists of high-performance parallel implementations of six common graph kernels, implemented in C++ and parallelized using OpenMP.

B. Graph Processors and Accelerators

The popularity of graph algorithms for analyzing big data has led to graph optimized processors and accelerators. The Cray Urika-GD graph processor [15] consists of multiple ThreadStorm/XMT CPUs, connected by a memory-coherent network, supporting multiple terabytes of memory. Motivated by the irregular memory accesses, there are no large caches, and each CPU supports 128 concurrent thread contexts to hide memory latency [16]. The more recent Urika-GX platform uses Intel Xeon nodes [1].

Song et al. [2] developed a graph processor based on sparse matrix operations. Ahn et al. [17] propose a processing-in-memory (PIM) solution for accelerating graph analysis applications. Ozdal et al. [18] propose a graph analytics accelerator for vertex-centric operations. Jin et al. [19] present their first steps in designing a dataflow based graph accelerator In this paper, we focus on general purpose processors (using the x86 instruction set), and analyze how these can be optimized for graph applications.

C. Graph Performance Analysis

A few papers discuss performance analysis of graph applications on commodity hardware. Beamer et al. [20] study the performance of common graph applications on an Intel Ivy Bridge server. They conclude that graph applications are memory bound, but they do not consume the full memory bandwidth. Furthermore, moderate cache locality exists and multithreading has limited benefit. Eisenman et al. [21] also study graph applications on the Ivy Bridge architecture. They find that prefetching is beneficial even for graph applications, and that using huge page sizes significantly improves performance. Liu et al. [22] compare the performance of graph applications on an Ivy Bridge CPU, an NVidia GeForce GPU and a Knight's Landing (KNL) Xeon Phi processor. They conclude that graph applications have a variable optimal thread count, that only a few applications benefit from the high-bandwidth MCDRAM memory, and that vector units are underutilized.

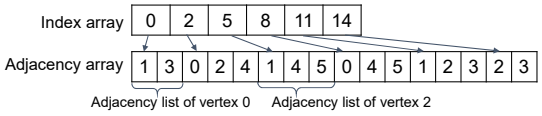


Fig. 1. CSR graph representation

All of these papers study the behavior of graph applications on (a) specific architecture(s) using performance counters. The goal of our paper is to discover insights into why this behavior occurs, to guide the design of future many-core architectures for graph analysis. Thereto, we complement hardware measurements with simulations that allow for a much deeper analysis on the root causes of certain observations, and that enable projecting graph application performance to future many-core designs.

III. Experimental Setup

We evaluate the GAP benchmarks [11], consisting of 6 common graph kernels: pagerank (pr), triangle count (tc), breadth first search (bfs), single source shortest path (sssp), connected components (cc) and betweenness centrality (bc). We only measure and simulate the actual parallel algorithm, excluding reading the graph from disk and preprocessing steps. As input graphs, we use the synthetic RMAT graphs from the graphchallenge website[1] with average degree 16 and scales 20, 22 and 24 ($\log_2$ of vertex count). We use synthetic graphs because their size can be controlled, which enables us to analyze the impact of graph size on performance. These graphs have a power law degree distribution, which is representative for many real-life graphs, such as social networks. Typically, there are a few vertices with many neighbors, and many vertices with few neighbors. Next to its representativeness, a power law degree distribution is more challenging to parallelize than for example a uniform distribution, because of the load imbalance it inherently causes.

All applications use the compressed sparse row (CSR) representation to store the graphs in memory, see Figure 1. Each vertex has an adjacency list, representing all other vertices it is connected to (its outgoing edges). To increase locality and avoid fragmentation, all adjacency lists are concatenated in one adjacency array (bottom array in Figure 1). The index array with size equal to the vertex count contains the index of the start of the adjacency list of that vertex.

We execute all applications on an 26 core Intel Skylake (SKX) processor (Xeon Platinum 8170) and a 64 core Intel Knight's Landing (KNL) processor (Xeon Phi 7250). To obtain more in-depth performance analysis data—such as CPI stacks through time, cache line efficiency and effective memory level

[1] http://graphchallenge.mit.edu

TABLE I
MACHINE AND CORE CONFIGURATIONS (OOO: OUT-OF-ORDER; IO: IN-ORDER)

	SKX	KNL	MSC
# cores	26	64	512
SMT/core	2	4	4
Execution pipeline			
type	OoO	OoO	IO
width (max IPC)	4	2	1
Frequency	2.4 GHz	1.4 GHz	1 GHz
L1 I cache	32 KB	32 KB	16 KB
L1 D cache	32 KB	32 KB	16 KB
L2 cache	1 MB	1 MB/2 cores	-
Shared L3	39 MB	-	-
Main memory	DDR4	MCDRAM	MCDRAM
Bandwidth	115 GB/s	460 GB/s	400 GB/s

parallelism (MLP)—we also simulate the applications using an in-house version of the Sniper multicore simulator [23] on similar SKX and KNL configurations.

Furthermore, we simulate a hypothetical 512-core configuration consisting of single-issue in-order cores (MSC: many small cores). The latter configuration is inspired by the Cray XMT/Threadstorm architecture [16], designed for graph applications, although with lower thread count per core (up to 4 instead of 128 threads). Note that each core is an independent core with its own instruction stream, which is different from a GPU, where instructions are shared between sets of cores. Because of the data dependent diverging code paths in graph applications, a many-core configuration is better suited than a warp scheduled architecture such as a GPU. All configurations support AVX512 vector instructions, to maximize compute and memory load bandwidth per core. The details of the configurations are in Table I.

All applications are compiled with the Intel icc compiler (version 17.0.4) using the -O3 optimization flag. The same binary is used for all experiments. Our simulator extends the public version of Sniper with IP-protected micro-architectural details of Intel processors, which cannot be released. However, the public version of Sniper is flexible enough to simulate similar architectures (out-of-order cores, in-order cores, heterogeneous configurations, cache hierarchy). Most of the analyses in this paper can be done with the public version of Sniper (e.g., CPI stacks) or require a few lines of additional code to collect (e.g., cache efficiency histograms).

IV. RESULTS

A. Vectorization of tc and cc

As we will show in the next sections, graph application performance is memory bound, due to the low compute to memory ratio. It is therefore important to maximize the number of in-flight memory operations and to exploit the memory bandwidth, as this reduces the memory overhead. Next to having more threads, per-thread memory parallelism can also be increased by using vector operations. In particular, Intel AVX512 can load 16 32-bit integers (the vertex id type for our setup) using a single instruction.

It is however not straightforward for the compiler to generate useful vector instructions from scalar source code, especially irregular graph code [22]. For the GAP benchmarks, only pagerank triggered automatic vectorization, using a gather instruction to get the pagerank values of 16 neighbors in one operation. We inspected the code, and found vectorization possibilities for triangle count and connected components. The other three applications (bfs, sssp and bc) are harder to vectorize, due to their expanding and contracting working sets.

For triangle count, we applied the vectorized list intersection algorithm proposed by Katsov [24], adapted to AVX512. For connected components, we added a `simd` pragma, to tell the compiler to ignore the potential cross-iteration dependencies. Due to the iterative algorithm, this operation is allowed. We also used the dynamic scheduling policy for OpenMP, instead of the standard static scheduling (the other applications already used dynamic scheduling).

The execution time on the KNL and SKX machines (hardware measurements) reduces by 60% for tc and 80% for cc. The number of instructions is halved for tc and quartered for cc, but the number of LLC misses remains about the same as in the scalar version. This means that the effective memory-level parallelism (MLP) has increased, which was the goal of our vectorization. In the remainder of the text, we show results for the vectorized applications.

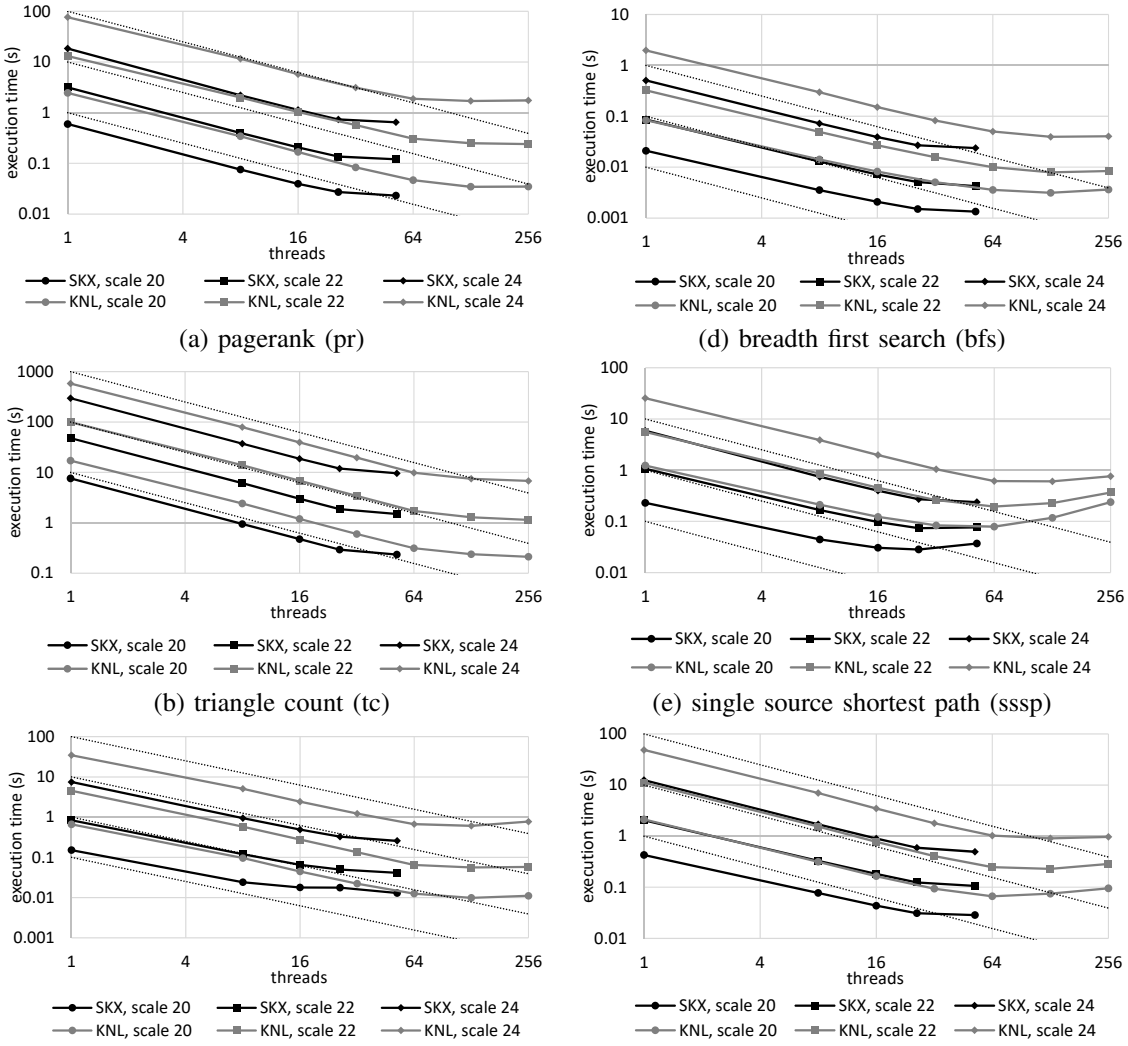
B. Performance Scaling on SKX and KNL

Figure 2 shows how each of the applications scale from 1 thread to the maximum thread count on SKX and KNL, measured on the actual hardware. Multiple SMT contexts (hyperthreading) are only used when thread count exceeds core count. The execution time as a function of thread count is shown in a log-log diagram, and some ideal scaling lines (dotted lines; execution time proportional to 1 over thread count) are added. If the scaling curve is parallel with a dotted line, it exhibits ideal scaling.

Most applications scale well up to the core count, even for the smallest input (scale 20). The applications do not profit much or even suffer from having more than 1 SMT thread per core (52 threads for SKX, and 128 and 256 threads for KNL). At equal thread count, SKX performs about 4 times higher than KNL, which is in line with the 2x wider pipeline, 1.7x higher frequency and the higher cache capacity (26×1 MB L2 and 39 MB L3 for SKX versus 32×1 MB L2 for KNL). The three benchmarks at the right column scale worse than those on the left column, especially for the scale 20 graph and when using SMT, where performance even degrades. This is because these applications have lower available parallelism, especially in the beginning of the execution, as we will show next.

C. Execution Profile

Figure 3 shows the CPI profile of selected applications, generated by Sniper. The graphs show CPI stacks over time, showing how much each event contributes to the total execution time. The bottom (dark) blue component is the time spent



(a) pagerank (pr)

(d) breadth first search (bfs)

(b) triangle count (tc)

(e) single source shortest path (sssp)

(c) connected components (cc)

(f) betweenness centrality (bc)

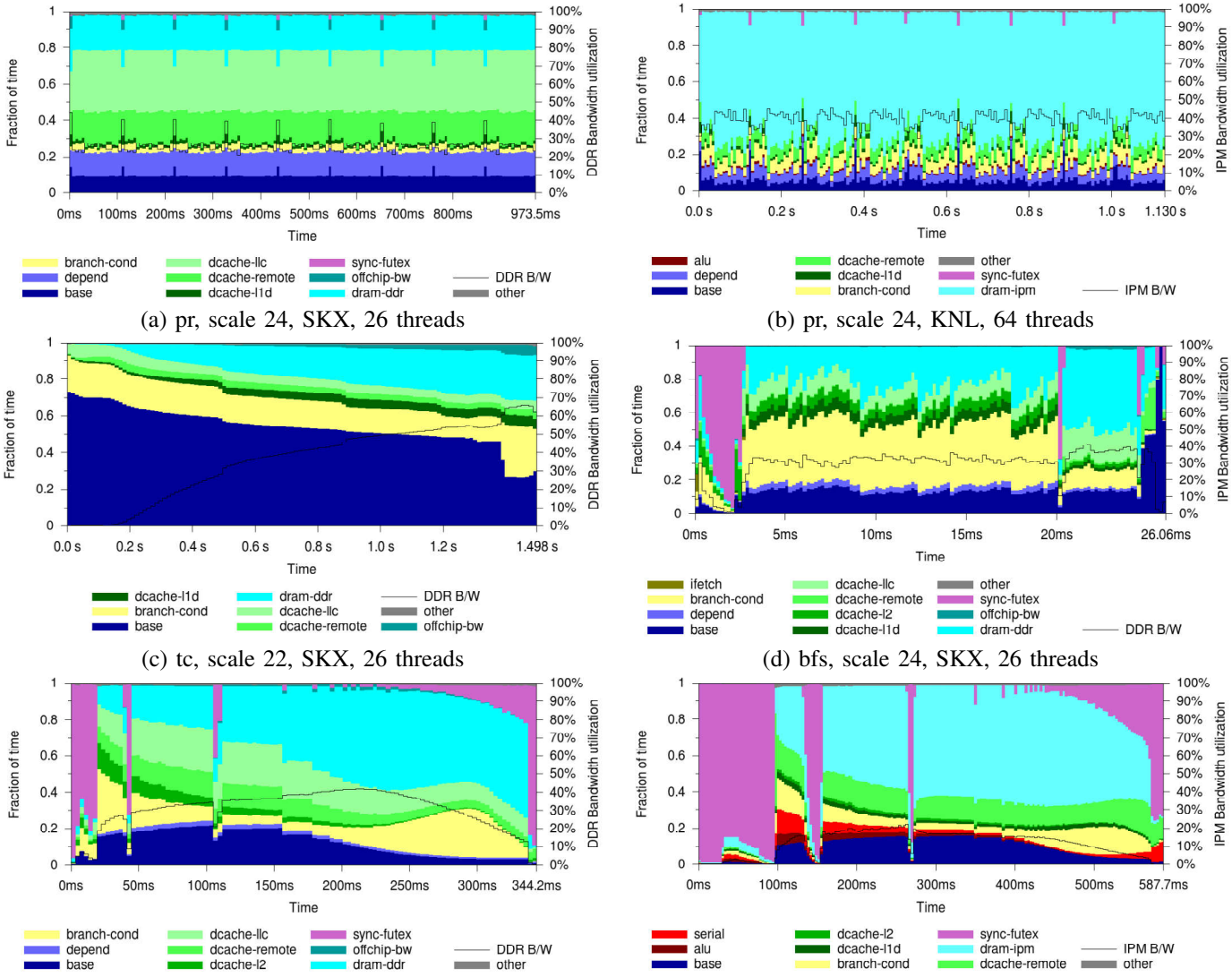
Fig. 2. Scaling behavior of graph applications on SKX and KNL. Log-log scale; dotted lines reflect ideal scaling.

in actually executing instructions. The lighter blue components shows the time spent waiting on dependences, and the yellow component the time spent in branch mispredictions. The green components reflect the time the core waits for cache accesses, and the top light and dark cyan part is the time waiting for DRAM (DDR or MCDRAM; the latter is called IPM in the graphs). The pink parts are idle cores due to synchronization, implemented in Linux as a futex. The black line shows the DRAM bandwidth usage over time.

The first two graphs are for pagerank, on SKX and KNL respectively. Connected components has a similar profile, so we omit it to save space. Performance for pr is dominated by memory accesses (green and cyan). On SKX, most of them are served by (remote) L2 caches or the L3 cache, which is why bandwidth consumption is low. Most of the cycles are spent in loading the pagerank value from neighboring nodes to calculate the new value for the current node. The size of this array of values is 34 MB, which means it can be kept in the shared L3 cache of 39 MB. The DRAM time is due to accesses to the adjacency lists, to find the vertex IDs of the neighbors. These do not fit in the cache (1.9 GB), but because they are kept in a linear structure (CSR representation, see Section III), they can make use of spatial locality and prefetching, reducing the number of misses to accessing the first element of each adjacency list.

The KNL has less cache capacity, so all accesses need to go to main memory, which is in this case the on-chip high-bandwidth MCDRAM. Bandwidth consumption is therefore much higher (note that the peak bandwidth of MCDRAM is 460 GB/s, so 40% utilization equals 184 GB/s). However, bandwidth consumption does not reach the maximum, despite the large amount of DRAM accesses. We will discuss the causes for this low bandwidth usage in the next section.

Triangle count (Figure 3(c)) is different from the other applications, as its base component is large (it has an IPC of 2). It also has the smallest memory component, though it increases near the end. Triangle count fetches the adjacency lists of two neighboring vertices, and looks for matches between these lists. These matches are common neighbors, creating



(a) pr, scale 24, SKX, 26 threads

(b) pr, scale 24, KNL, 64 threads

(c) tc, scale 22, SKX, 26 threads

(d) bfs, scale 24, SKX, 26 threads

(e) sssp, scale 24, SKX, 26 threads

(f) sssp, scale 24, KNL, 64 threads

Fig. 3. CPI stack profile of selected applications

a triangle with themselves. Finding matches is a compute-intensive task, even if all lists are sorted. As the graph is sorted with decreasing degree, the lists for the first nodes are long, having a high compute to memory ratio. As lists become shorter (often size 1 at the end), the amount of calculation is smaller, leading to more memory bound behavior. The high compute fraction is also an explanation why SMT has no benefit: the functional units are heavily used by one thread, so there is no benefit of adding more threads to increase functional unit occupation.

The next three graphs have a clear fraction of synchronization penalties (pink part). The organization of these application (bfs, sssp and bc (not shown because similar to the other two)) is similar: they start from a single source vertex, and then build up a queue with all its neighbors, continuing with their neighbors, and so on. In the beginning, there is not much parallelism (as the queue length is small), and as the processing of each front ends with a barrier, there is potential load imbalance at the end of each step. This explains their inferior scaling behavior: as thread count increases, the fraction of synchronization penalty increases (compare Figure 3(e) and (f): from 26 threads on SKX to 64 threads on KNL, the area in pink increases). It also explains why SMT degrades performance: the longer running threads, which in the end determine total performance, execute slower because of sharing core resources with the co-running thread(s).

Breadth first search (Figure 3(d)) has a particularly high branch miss component, although all other applications also have a clearly visible branch component. Graph applications have a large fraction of hard-to-predict branches due to the irregularity of graphs. The branch with the largest penalty for bfs is checking whether a vertex neighboring the current front has been visited before. Because an edge can occur between any pair of vertices, this branch is hard to predict, and will switch from highly likely in the beginning of the execution (when few vertices have been visited) to unlikely near the

end. At the very end, the base component gets big, and the branch component disappears. This is due to an optimization in the code: if the non-visited vertex count is low, it looks for breadth first search tree parents of the non-visited vertices, instead of looking for children of the already visited vertices. This leads to more efficient code, because fewer vertices need to be checked, and the probability that a vertex has a parent that belongs to the current front is high.

D. Memory Bandwidth Usage

All applications, except for tc, are memory bound, indicated by large cache (green) and DDR/IPM (cyan) components. Pr on SKX (Figure 3(a)) has a working set that fits into the L3 cache, which explains why bandwidth consumption is low. All other applications have a working set that is too large for L3, indicated by a significant DDR/IPM component. Their main bottleneck is waiting for main memory accesses. The penalty of memory accesses can be reduced by exploiting more memory-level parallelism (MLP) and using more of the available memory bandwidth. However, the memory bound applications have a bandwidth usage that is lower than 50% of the available bandwidth (black line in the CPI stack profiles), and even less than 15% for bfs, sssp and bc on KNL, which means that they are not effective in exploiting MLP. To find the cause for this low bandwidth usage, we use Little's law, stating that the bandwidth consumption equals the number of outstanding requests (in bytes) divided by the average latency of a single request:

$$\text{BW} = \frac{\text{outstanding B}}{\text{latency}} \tag{1}$$

$$= \frac{(\text{LLC misses} + \text{prefetches}) \times \#\text{threads} \times \text{cache line B}}{\text{access} + \text{contention latency}} \tag{2}$$

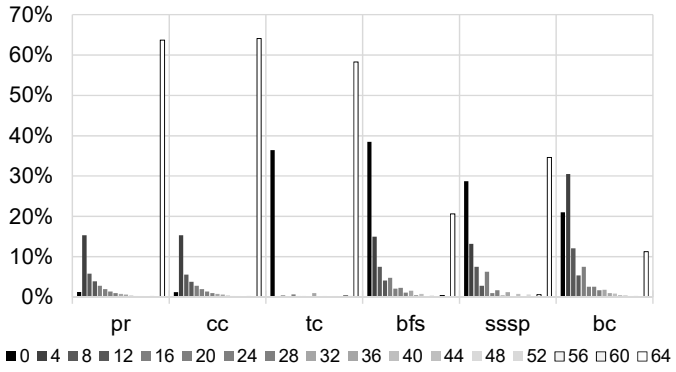
The number of outstanding requests equals the number of outstanding cache misses and prefetches per thread times the number of threads times the cache line size (64 byte). Outstanding cache misses per thread is related to the cache miss rate, the available MLP in the application and the number of concurrent accesses supported by the caches (the number of miss status handling registers or MSHRs). All applications, except tc, have a low amount of prefetches, because of the irregular memory pattern that cannot be predicted by a stride stream prefetcher. Tc has a more regular access pattern, because it only loads adjacency lists and it has no per-vertex data array that is indirectly accessed. The higher bandwidth consumption at the end of the execution (Figure 3(c)) is mainly caused by prefetches, which explains why the performance impact of bandwidth saturation (the offchip-bw component) is relatively small.

Bfs, sssp and bc have a relatively low amount of concurrent misses compared to the other applications: Sniper reports an effective MLP of 1 to 4, compared to 10 to 12 for the other applications (there are 12 MSHRs on the DL1 cache). The reason is that they use atomic operations (atomic add or compare-and-swap). Atomic operations cannot be vectorized, and they are also serialized to ensure their atomicity, limiting the MLP. Furthermore, these applications also have regions with low thread count, which limits the attainable bandwidth consumption.

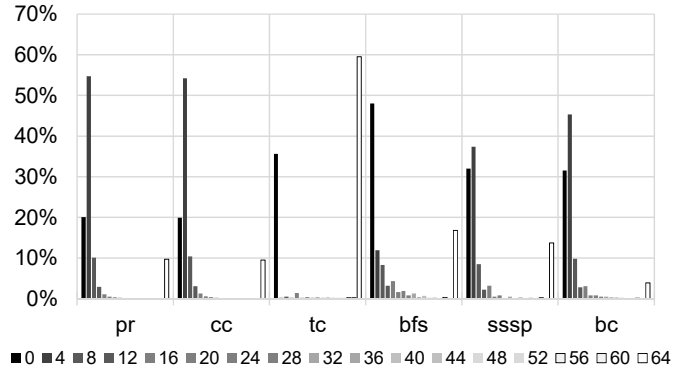
Finally, long latency accesses also limit the exploitable bandwidth, which is what occurs for example for pr on KNL (Figure 3(b)). The average load-to-use latency for an MCDRAM access (L2 miss) is 190 ns, which is composed of the DL1, L2, tag directory and MCDRAM access time (150 ns), TLB miss penalty and network-on-chip contention. The large amount of TLB misses is caused by random accesses to the 1.9 GB adjacency list structure, whose address range does not fit in the TLB. With an average MLP of 10, running on 64 cores and 64 bytes per load, this results in a maximum bandwidth consumption of $\frac{10 \times 64 \times 64}{190} = 213$ GB/s, or 46% of the peak bandwidth. In fact, maximum bandwidth consumption can only be reached by issuing enough prefetches next to the core demand misses. However, the irregularity of the access stream prevents issuing useful prefetches. The already high demand miss MLP explains why SMT does not help performance: the number of outstanding requests is close to maximum (10 out of 12) and does not increase with more threads, so bandwidth and performance does not improve.

Because most applications are memory bound (except tc and pr on SKX), increasing bandwidth utilization improves performance. A few recommendations to increase bandwidth utilization are:

- Increase the available MLP per thread: avoid atomic operations, or implement vector versions of atomic operations.
- Increase the number of threads: can be done by increasing core count or more threads per core through SMT. However, SMT does not work if a thread already uses (close to) the maximum outstanding misses that the hardware allows.
- Increase the parallelism: increasing thread count does not help if there is not enough parallelism. Algorithms should be revised to increase the exploitable parallelism, certainly if even more cores are used. For example, more parallel speculative work could be done (e.g., by removing thread barriers), at the cost of needing more iterations to reach stability. Another option is to use edge-centric algorithms instead of the common vertex-centric algorithms. Because the compute time per edge is more constant than the compute time per vertex (which often depends on the degree), edge-centric algorithms should show less load imbalance.
- Decrease the latency per access: reduce the load-to-use path for cache misses; reduce contention in shared resources and networks; reduce TLB misses by using larger page sizes.
- Implement prefetchers that are able to prefetch indirect memory streams, for example the list prefetcher implemented in the IBM Blue Gene/Q processor [25] or the indirect memory prefetcher proposed by Yu et al. [26].



(a) SKX, 26t



(b) KNL, 64t

Fig. 4. Cache line usage (in bytes) histograms on SXK and KNL; the Y-axis is the fraction of fetched 64 byte cache lines that provide 0, 4, 8, etc. useful bytes to the CPU.

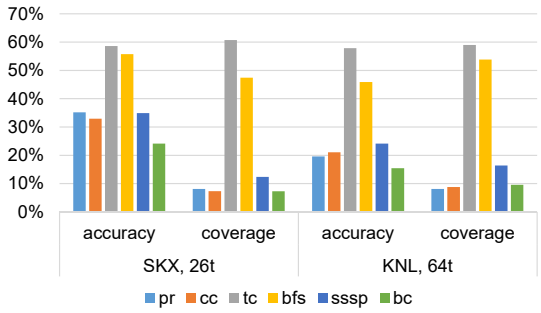


Fig. 5. L2 prefetcher accuracy and coverage.

E. Cache Line Efficiency and Prefetching

Figure 4 shows histograms of the cache line efficiency, defined as the number of bytes used (read and/or written) by the CPU of each 64-byte cache line fetched from main memory (DDR or MCDRAM). In other words, of all cache lines fetched from memory (by demand or prefetched), how many are fetched uselessly (0 used bytes), and how many provide 4, 8, 12, etc., up to the full 64 bytes to the core before they are evicted. These fractions are collected by Sniper during simulation. A low useful fraction means inefficient use of cache capacity and memory bandwidth, while a high fraction means that the application profits from spatial locality.

For most applications, there is a large fraction of cache lines where no byte is used (leftmost '0' column per application). This is due to useless prefetches: the data is brought in by the prefetcher, but it is never used. Figure 5 shows the accuracy and coverage of the prefetcher at the L2 cache. Accuracy is defined as the number of useful prefetches relative to the total number of prefetches, and coverage is the number of avoided cache misses (i.e., useful prefetches) relative to the total number of misses in absence of a prefetcher (i.e., number of misses with prefetcher plus the number of avoided misses). For all applications except tc and bfs, accuracy and coverage is relatively low. Most of the memory accesses for these applications have an irregular access pattern to a large structure that does not fit into cache, which explains their low accuracy. For tc and bfs, most of the accesses are to the adjacency lists of the vertices. These are stored consecutively in memory, so for long adjacency lists, the prefetcher performs well. However, most vertices have only a few neighbors, causing the prefetcher to fetch too far, which is why even for tc and bfs, accuracy and coverage is less than 60%.

This analysis seems to suggest that prefetching is useless and may hurt performance (except for tc and bfs). To test this statement, we execute all applications on the SKX and KNL hardware with disabled hardware prefetchers. We find that performance improves only for pr on KNL (20% higher performance). For all other applications and on SKX, we see performance degradations between 10% and 25%, and up to 45% for tc and bfs. Prefetching can hurt performance if it causes memory bandwidth contention and/or wastes cache capacity that could be used more effectively. Because bandwidth consumption is low and the memory access pattern is not cache-friendly, useless prefetching does not hurt performance. Therefore, even a 10% reduction in misses due to prefetching (coverage) has a positive impact on performance. For pr on KNL, disabling prefetching increases the L2 hit rate for accesses to the pagerank value array. The combined L2 cache capacity is 32 MB (32 times 1 MB), which is able to hold most of the 34 MB array. When prefetching is enabled, adjacency lists are prefetched, but often too far, evicting useful parts of the pagerank value array.

Looking back to Figure 4 and disregarding the zero column, we notice two peaks: one at 4 byte usage (except for tc) and one at 64 byte usage (i.e., the full cache line is used). This bimodal distribution can be explained by the data structures used by the applications. There are two types of structures: one that contains per-vertex data (e.g., pagerank values, bfs tree parents, or cluster IDs) and the adjacency list array. The per-vertex arrays are accessed in an irregular way, because they are often indexed by the list of neighbors of a vertex, which has no locality. Therefore, only one element (4 byte) of that cache line is used before it is evicted. On the other hand, adjacency lists are stored in a consecutive way, meaning that there is high spatial locality, and all elements of a cache line are useful. Tc has no per-vertex data, the only data structure is the CSR, which explains the absence of 4-byte cache line usages.

We can make the following recommendations:

- Current hardware prefetchers are mostly beneficial for performance, but they generate many useless prefetches, which could be a problem if memory bandwidth and/or cache capacity are contended. As suggested in the previous section, an indirect memory prefetcher could largely increase the accuracy and coverage of the prefetchers.
- There are two different memory access patterns: one without locality, using only 4 bytes of a cache line, and one with high spatial locality (adjacency list array). If memory bandwidth and/or cache capacity are contended, and if there is a mechanism to discern these two patterns (either in software or through profiling in hardware), the no-locality stream can be served by non-cached 4-byte memory accesses, releasing cache capacity and memory bandwidth (by fetching only 4 bytes instead of a full 64 byte cache line).

F. Extrapolation to Many Small Cores

Previous work (see Section II) on graph application performance analysis reports two challenges for graph applications: their memory-boundedness and the irregular access pattern. This has inspired graph processor architects, such as the Cray XMT architecture [16], to opt for a many small core design with high thread count per core. To complete our analysis, we simulate the applications on a hypothetical future many small core (MSC) architecture, detailed in Table I. Because most of the time is spent in waiting for memory, there is no need for powerful out-of-order cores, so more energy-efficient in-order cores can be used. The irregular access pattern is not fit for caching and prefetching, so there are no L2 and shared L3 caches, in favor of high-bandwidth memory to serve many concurrent memory requests. In order to saturate this high memory bandwidth with demand misses, we simulate 512 cores on one chip, which is possible because of the small cores and caches. Additionally, having multiple threads per core further increases the throughput and hides the latency of the individual accesses. Because we notice diminishing returns and even performance degradations at 4 threads per core, we decided not to simulate more than 4 threads per core.

Figure 6 shows the simulated execution time of the applications on this architecture for 64 to 2048 threads. SKX and KNL execution times (measured on hardware) are also shown for reference. Figure 7 shows the CPI stack profile for some applications on MSC at 512 threads. A lot of interesting conclusions can be drawn from these graphs.

First, we compare the performance of KNL and MSC at 64 threads. For most applications, the performance of MSC is only slightly worse than that of KNL: most applications are memory bound, which means that the out-of-order pipeline of the KNL cores is stalled most of the time. One exception is tc, where MSC at 64 threads is more than 5 times slower than KNL. Tc is compute bound, see Figure 3(c), so performance is hit by using a single-issue in-order pipeline compared to a 2-wide out-of-order pipeline. Furthermore, tc benefits from prefetching (see Figure 5) and caching, which is absent in the MSC architecture.

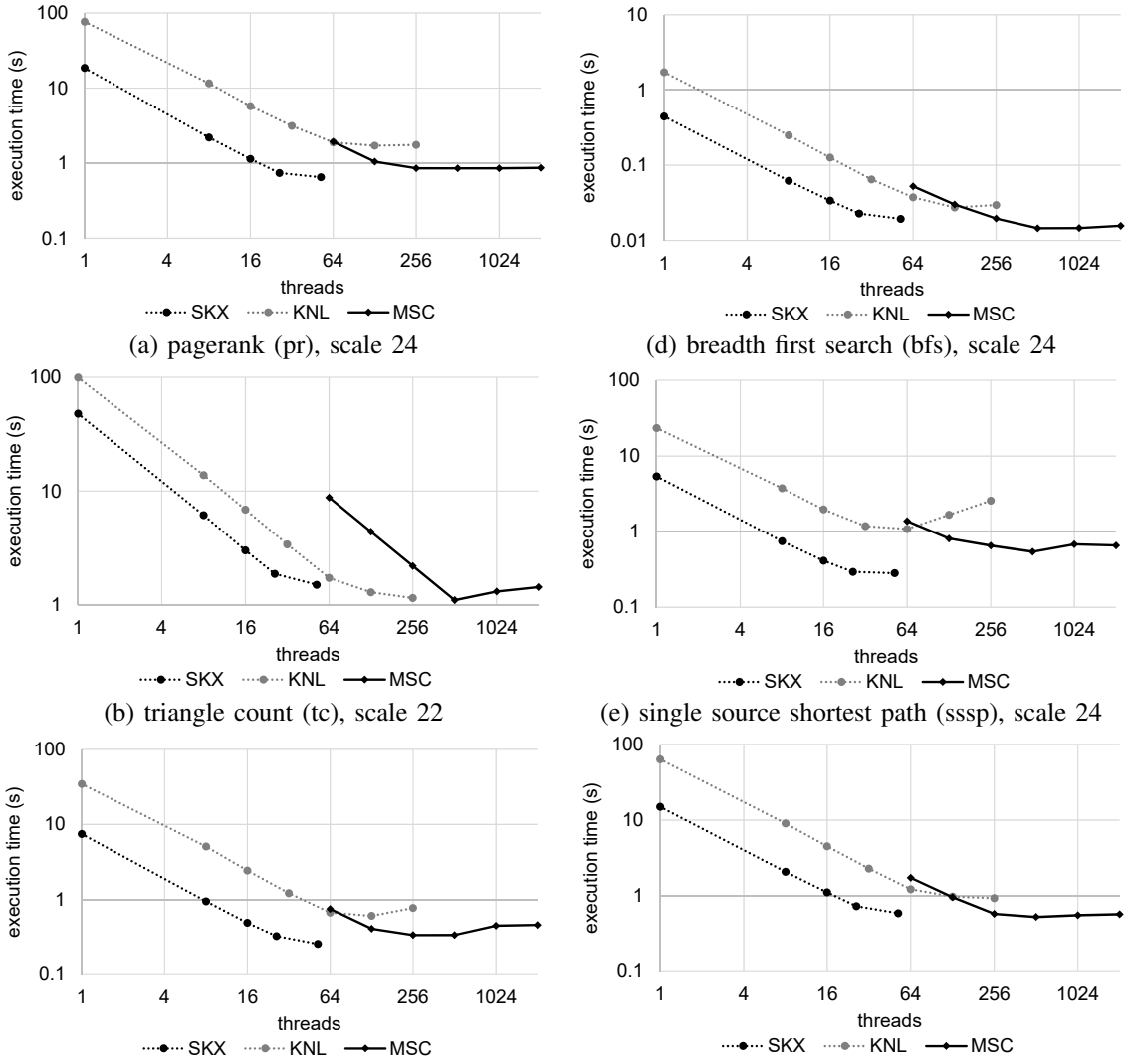
For pr and cc, performance scales well until 128 threads (using only 1/4 of the cores), after which performance saturates. An analysis shows that from 256 threads, memory bandwidth becomes the main bottleneck, see Figure 7(a). In order to improve the scaling of pr and cc, either the available memory bandwidth needs to increase or bandwidth-saving techniques should be implemented, such as the 4-byte accesses discussed in the previous section. For example, 90% of the DRAM accesses for pr contain 4 or fewer useful bytes. Converting them to 4 byte accesses reduces the number of bytes loaded from DRAM by 85%. Figure 7(a) suggests that reducing bandwidth contention might decrease the execution time by as much as 80% (the size of the light cyan component). Note that neither KNL nor MSC is able to beat SKX, despite the lower core count of SKX. This is because of the large shared L3 cache of SKX, which is able to hold most of the data and has a lower latency and larger bandwidth than MCDRAM.

Tc continues to scale until 512 threads, because of its low bandwidth requirements. However, because its per core performance is much lower than that of KNL, 512 threads are needed to beat KNL and SKX. This application clearly profits from more powerful cores and caching. Using multiple threads per core severely hurts performance. The reason is that vertices are sorted by their degree, and the first thread processes the vertex with the highest degree. Because that degree is much higher than that of the other nodes (power law distribution), and the process time is quadratical in the degree, this thread usually lasts until the end of the application, while the other threads process all other vertices. When multiple threads execute on one core, each thread has lower performance than when it runs alone on a core, especially for compute bound applications such as tc. Therefore, the first thread is slowed down, making it run longer. Near the end of the application, it is the only thread running, and the application has to wait until that thread has finished.

Bfs, sssp and bc stop scaling well at 256 threads. Although they also suffer from bandwidth contention at some points in their execution (see Figure 7), their main issue is the limited parallelism. The beginning of the application uses only a single thread, whose performance is lower than a single thread on KNL, extending the single-threaded execution phase. The high thread and core count is only beneficial for small parts of the application, which does not compensate for the large parts where active thread count is low.

We make the following recommendations for a graph many-core processor:

- Except for tc, the choice of in-order cores is appropriate. However, for some applications, it may be beneficial to keep some of the high-performance core features, such as prefetchers and a moderate L2 cache.
- Because of limited parallelism, it can be beneficial to include a few high-performance cores, i.e., a heterogeneous design. If thread count is low, execution can be sped up by executing these threads on the high-performance cores, alleviating the Amdahl bottleneck.



(a) pagerank (pr), scale 24

(d) breadth first search (bfs), scale 24

(b) triangle count (tc), scale 22

(e) single source shortest path (sssp), scale 24

(c) connected components (cc), scale 24

(f) betweenness centrality (bc), scale 24

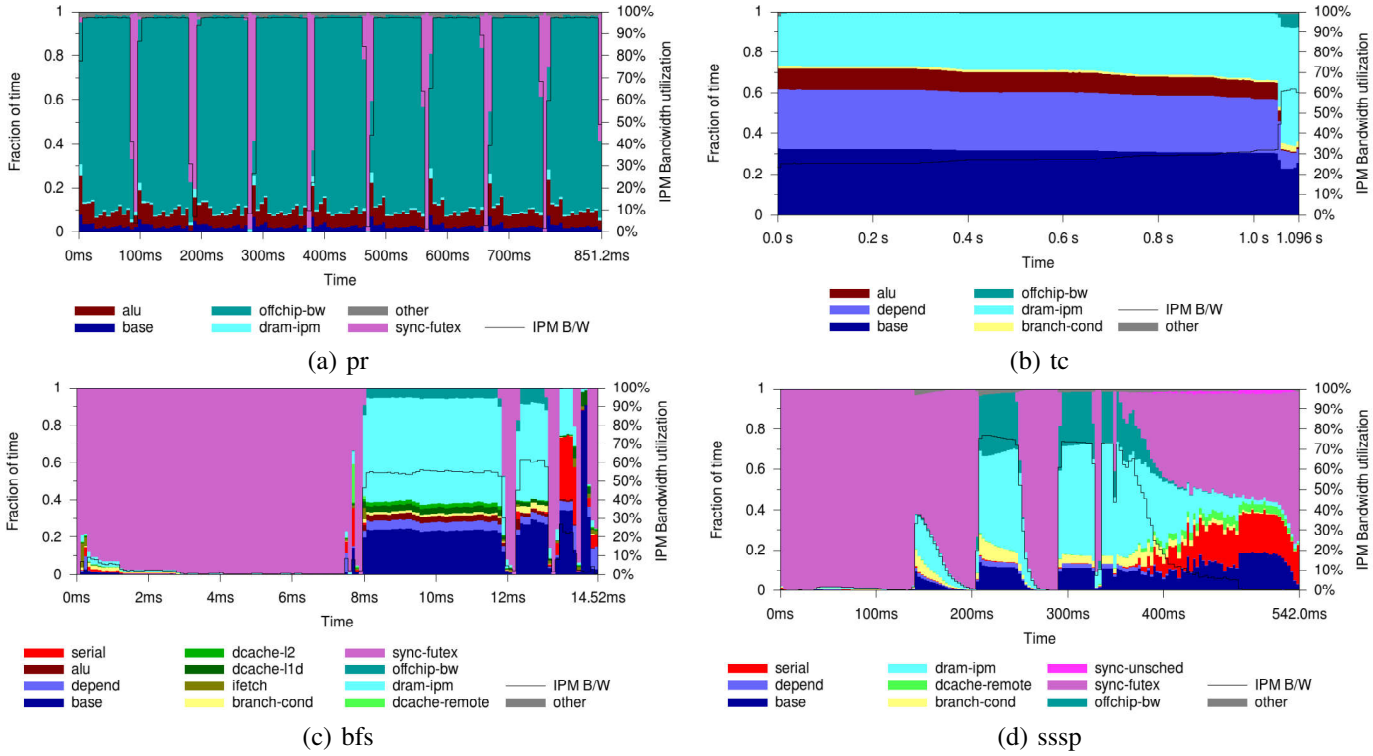
Fig. 6. Scaling behavior of graph applications on SKX, KNL and MSC (log-log scale)

- All of the other recommendations from the previous sections are still valid: increase software parallelism, enable selective subline DRAM accesses, include dedicated prefetchers, etc.

G. Heterogeneous Many-Core

Phases with low active thread count can be accelerated by executing them on powerful 'big' cores. We therefore simulate a heterogeneous many-core configuration, where in the MSC configuration, we replace 64 small cores with 4 SKX cores (we verified that 1 SKX core is approximately area-equivalent with 16 small in-order cores). Note that this heterogeneous configuration has the same ISA in all cores, and all threads perform the same algorithm (homogeneous threads). Therefore, the initial mapping of the threads on big or little cores has less impact than for heterogeneous multithreaded applications and/or functionally heterogeneous architectures (e.g., CPU and GPU). The scheduler is adapted to maximize the usage of the big cores: when a thread that is executing on a big core is stalled, another thread is 'stolen' from a small core to execute on that big core. When there are only a few active threads, this ensures that these are executed on the big cores, speeding up the phases with low thread count. Note that all applications use the OpenMP dynamic schedule, so the threads on the big core only stall when there is no work left for the current iteration. This means that thread migrations occur infrequently and only at the end of each iteration, minimizing the overheads of thread migration, such as cache warming and NUMA effects (which are modeled in Sniper).

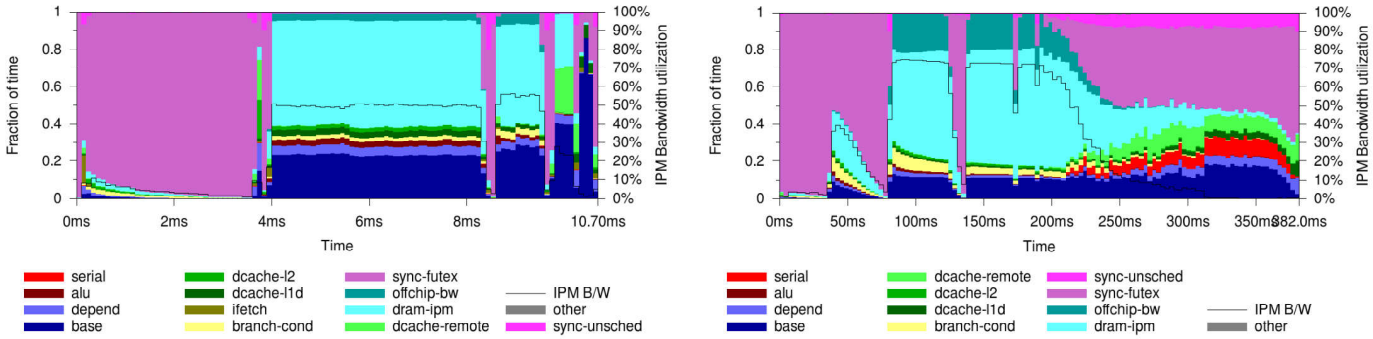
Figure 8 shows the CPI stacks for bfs and sssp on the heterogeneous configuration (compare with Figure 7(c) and (d)). We indeed notice a significant execution time reduction in the phases with low parallelism, while the parallel phases are slightly longer because of the lower overall thread count (452 threads instead of 512 threads). Overall, the execution time reduction is moderate, indicating that we can not rely just on heterogeneity to solve the imbalance problem, and that



(a) pr (b) tc

(c) bfs (d) sssp

Fig. 7. CPI stack profile on MSC with 512 threads of selected applications



(a) bfs (b) sssp

Fig. 8. CPI stack profiles on the heterogeneous configuration.

increasing parallelism in software is still needed.

V. Conclusions and Recommendations

Our analysis of graph analysis applications on many-core architectures has confirmed prior insights about the irregularity of these applications, but it has also revealed novel insights that guide the development of a future many-core graph processor. We make the following recommendations:

- Implement two different memory access instructions: one that exploits locality and regularity by using caching and prefetching; and one that does not use caches or prefetchers, and that fetches only 4 or 8 bytes from memory instead of a full cache line. This leads to a more efficient cache and memory bandwidth usage.
- Implement many cores to maximize the number of outstanding misses and saturate high-bandwidth memory. This needs to be combined with a limitation of the memory access latency to further increase memory bandwidth usage, e.g., by eliminating TLBs and limiting the number of cache levels. In our setup, adding more threads per core (SMT) is not very beneficial and can even hurt performance.
- In order to be able to efficiently use that many cores, the algorithms need to be revised to increase parallelism and decrease load imbalance.
- As not all load imbalance can be removed, add a few high-performance cores to handle phases with low parallelism.

We conclude that high-performance highly-parallel graph analysis is a highly relevant and ongoing research and development topic.

REFERENCES

[1] Cray, "Urika GX," 2017. [Online]. Available: https://www.cray.com/sites/default/files/Cray-Urika-GX-Technical-Specifications.pdf

[2] W. S. Song, V. Gleyzer, A. Lomakin, and J. Kepner, "Novel graph processor architecture, prototype system, and results," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–7.

[3] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the GraphBLAS," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–9.

[4] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, 2011, pp. 12–25.

[5] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 14, 2014, pp. 599–613.

[6] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[7] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," in *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence (UAI2010)*, 2010.

[8] J. Seo, S. Guo, and M. S. Lam, "SociaLite: Datalog extensions for efficient social network analysis," in *IEEE 29th International Conference on Data Engineering (ICDE)*, 2013, pp. 278–289.

[9] Apache, "Apache giraph," 2013. [Online]. Available: http://http://giraph.apache.org/

[10] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 979–990.

[11] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[12] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "GraphBIG: understanding graph computing in the context of industrial solutions," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2015)*, 2015, pp. 1–12.

[13] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *2015 IEEE International Symposium on Workload Characterization (IISWC)*, 2015, pp. 44–55.

[14] S. D. Pollard and B. Norris, "A comparison of parallel graph processing implementations," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 657–658.

[15] Cray, "Urika GD," 2014. [Online]. Available: https://www.cray.com/sites/default/files/resources/Urika-GD-TechSpecs.pdf

[16] A. Kopser and D. Vollrath, "Overview of the next generation Cray XMT," in *Cray User Group Proceedings*, 2011, pp. 1–10.

[17] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, 2015, pp. 105–117.

[18] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16, 2016, pp. 166–177.

[19] H. Jin, P. Yao, X. Liao, L. Zheng, and X. Li, "Towards dataflow-based graph accelerator," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 1981–1992.

[20] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an Ivy Bridge server," in *2015 IEEE International Symposium on Workload Characterization (IISWC)*, 2015, pp. 56–65.

[21] A. Eisenman, L. Cherkasova, G. Magalhaes, Q. Cai, and S. Katti, "Parallel graph processing on modern multi-core servers: New findings and remaining challenges," in *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2016, pp. 49–58.

[22] X. Liu, L. Chen, J. S. Firoz, J. Qiu, and L. Jiang, "Performance characterization of multi-threaded graph processing applications on Intel many-integrated-core architecture," in *Proceedings of the International Symposium on Performance Analysis of Software and Systems (ISPASS 2018)*, 2018.

[23] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 28:1–28:25, Aug. 2014.

[24] I. Katsov, "Fast intersection of sorted lists using SSE instructions," 2012. [Online]. Available: https://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/

[25] I. Chung, C. Kim, H.-F. Wen, G. Cong *et al.*, "Application data prefetching on the IBM Blue Gene/Q supercomputer," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, p. 88.

[26] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015, pp. 178–190.