Cache Line Sharing and Communication in ECP Proxy Applications

Joshua Randall(✉), Alejandro Rico, and Jose A. Joao

Arm Research, Austin, TX, USA
{joshua.randall,alejandro.rico,jose.joao}@arm.com

Abstract. Scientific computing codes rely on efficient parallelization to achieve performance. This parallel efficiency is reduced by factors such as communication, serialization, and data sharing. In this work, we examine interactions between OpenMP threads in the context of a Chip-multiprocessor (CMP). We first analyze cache line sharing to observe how often multiple threads are accessing the same data. We then look at producer-consumer and write-invalidation interactions between these threads. These interactions are implemented with cache coherence operations and demonstrate interference between threads. We find that none of the codes studied show prohibitive amounts of communication and many interactions between threads follow simple patterns. Our work discovers opportunities to increase parallel efficiency in the analyzed codes and provides motivating data for research into CMP design.

Keywords: Cache-communication · Coherence · Multi-core · Performance analysis · Scalability

1 Introduction

Multi-core processors with an increasing number of cores have potential to significantly boost performance of parallel applications, including high-performance computing (HPC) codes, by running multiple MPI processes and OpenMP threads in parallel. However, that potential may be thwarted by inter-thread communication, which can reduce single thread performance by disrupting cache locality. We identify two examples of inter-thread communication.

First, *producer-consumer communication* happens when one thread (producer) writes data that another thread (consumer) reads through a cache-to-cache transfer from the producer private cache to the consumer private cache. Second, *write invalidation communication* is when one thread running on core A writes to a cache line that is held in one or more remote private caches. These remote caches must be invalidated before the cache line can be brought in exclusive state and written to in core A's cache. Write invalidations can occur due

This work was in collaboration with Cray and funded in part by the DOE ECP PathForward program.

© Springer Nature Switzerland AG 2019
X. Fan et al. (Eds.): IWOMP 2019, LNCS 11718, pp. 306–319, 2019.
https://doi.org/10.1007/978-3-030-28596-8_21

to writes to truly shared data or due to writes to thread-private data that is on different words within the same cache line, which is called *false sharing*.

Application developers may improve parallel performance by reducing inter-thread communication. False sharing can be eliminated by allocating shared data and private data for different threads on different cache lines through alignment and padding. However, producer-consumer communication and write invalidations of truly shared data is intrinsic to the algorithm and can only be avoided with algorithmic changes.

In this paper, we study OpenMP inter-thread communication of HPC Proxy-Apps with a characterization of the following interactions:

- **Cache line sharing** among OpenMP threads to understand how inter-thread code and data sharing occurs on the cache hierarchy.
- **Producer-consumer communication** that results in direct cache-to-cache transfers.
- **Write invalidation communication** that occurs when shared data is modified.

Frequency of communication interactions indicate their likelihood of impacting performance and scalability of the applications, while interaction patterns visualize data movement between cores and provide insight into possible data movement optimizations.

2 Experimental Setup

2.1 ECP Proxy Apps

This Exascale Computing Project (ECP) [5] provides a collection of proxy applications that demonstrate a variety of multi-threading characteristics from HPC codes.

These proxy applications model characteristics of large scale HPC codes without the large code bases and problem sizes that are inherent to production HPC codes. These miniaturized codes enable detailed analysis of how these HPC codes run on single nodes or clusters. For our analysis, we examined behavior of these proxy apps from the perspective of a single CMP. Specifically, we evaluated the coherence behavior that these proxy apps demonstrate as the number of OpenMP threads increases.

Two of the proxy applications we evaluate, CoMD and miniFE, are no longer part of the latest release of the proxy application suite, but are still interesting to software developers. LULESH [7] is not part of the ECP proxy application suite, but has been a widely studied proxy app in multiple DOE exascale initiatives.

Inputs and Scaling. Table 1 shows the scaling strategy and base input sets used in this paper. Weak scaling, i.e., scaling problem sizes proportionally with the number of threads, was used when possible, in order to keep the amount of data per thread constant. For AMG, CoMD, ExaMiniMD, and LULESH,

we maintained a cubic input size and doubled the volume as thread counts doubled. Therefore, doubling threads scaled each dimension by a factor of the cube root of 2. For miniFE, we maintained a constant z dimension and alternated doubling the y and x dimensions as thread counts doubled. For the other proxy applications, we applied strong scaling, where we problem size remains the same when increasing the number of threads.

Table 1. Inputs and scaling for proxy applications

Proxy App	Scaling Used	Parameters (2 threads)
ExaMiniMD	Weak	50 50 50
AMG	Weak	-n 94 94 94 -P 1 1 1
miniFE	Weak	-nx 32 -ny 16 -nz 128
LULESH	Weak	-s 25 -i 10
CoMD	Weak	-e -x 20 -y 20 -z 20 -T 4000 -N 2 -n 1
miniAMR	Strong	--nx 16 --ny 16 --nz 16 --num_vars 40
SWFFT	Strong	2 512
XSBench	Strong	-t 2 -l 5000000 -s large
miniVite	Strong	-n 150000

2.2 DynamoRIO

We measured data accesses and data sharing of the proxy apps using DynamoRIO [4]. DynamoRIO is a dynamic binary instrumentation tool that includes a cache simulator. This tool does not include a detailed core model, so it does not simulate cycles and timing, but it can produce an accurate estimation of cache behavior. While multithreaded simulation is supported in DynamoRIO, we had to implement coherence support on top of the latest open source version to properly track cache line sharing. Our results were collected during the parallel phase of execution for each proxy application. We statically mapped one logical thread per core in our simulations.

2.3 Compiler and Runtime System

All proxy applications were compiled using GCC version 7.1.0 and memory traces were gathered for AARCH64 code running the libgomp OpenMP runtime included with GCC. All of the proxy applications use OpenMP with the exception of ExaMiniMD, which is parallelized using Kokkos. We measured cache line communication during the entire parallel execution phase of each proxy apps.

2.4 Evaluation of Cache Line Sharing

We simulated three levels of cache in DynamoRIO. A shared LLC with 2 MB per core backs up coherent 512 KB private L2 caches, which are inclusive with 64 KB L1I and L1D caches. Our simulated cache hierarchy uses a directory-based write-back cache policy to keep the L2 caches coherent. Each L2 and its child L1 caches perform accesses for a single thread. We measured the sharing state of unique cache lines between all L2 caches over time, as well as how widely each of these cache lines was shared between L2 caches. We also measured the frequency with which each private cache shares data with each other private cache.

2.5 Evaluation of Inter-thread Communication

Producer-consumer communication may be analyzed by tracking reads and writes at a word granularity. This analysis would be hardware agnostic and may not reflect the communication that actually occurs between cores during execution. We choose to analyze communication *coherently* to qualify communication that manifests in inter-cache transactions. In this context, communicating reads are remote accesses to dirty cache lines, or lines that have been written to and not evicted from the writing core's private cache. This analysis will show actual movement of data from communicating accesses between private caches during execution, and will include the effects of *false communication* caused by *false sharing*. False communication refers to unnecessary communication between caches that are accessing different words in the same cache line. Our results show the frequency of coherent communication during execution and reveal patterns in this communication.

Write invalidations occur when a core writes to a cache line of which another copy exists in another core's cache. The writing thread must complete an invalidation of cache line copies in any other cache before the write may be completed. Writing to data that is widely shared will add latency for the write operation and increase traffic in the Network-on-Chip (NoC). We counted write invalidations during execution of the proxy applications to compute their frequency. We also observe any patterns between frequently writing threads and frequently invalidated threads.

3 Results and Discussion

3.1 Cache Line Sharing Analysis

Cache line sharing occurs when multiple threads read from a cache line within the same period of time, causing the copies of that cache line to exist in multiple private caches simultaneously. This analysis offers insight into how well data is isolated between threads and how often different threads are operating on the same or adjacent data.

Figure 1 shows the number of shared caches lines as a percentage of total L2 capacity. We sampled caches at equal intervals during the parallel phase of

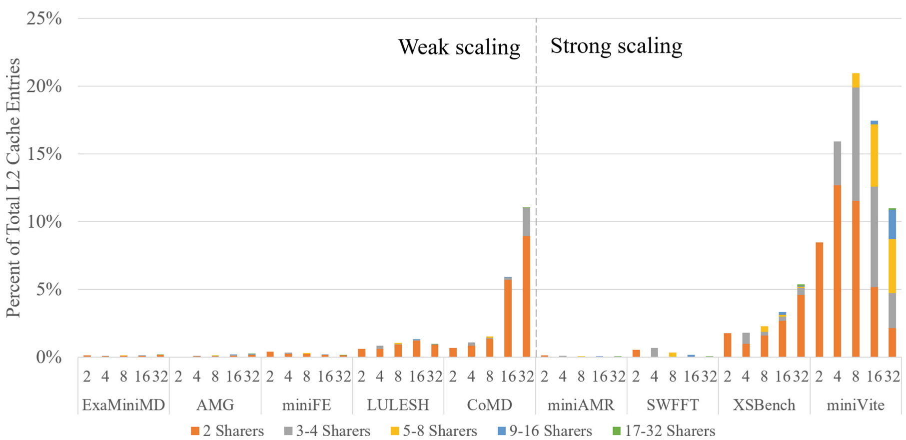

Fig. 1. Percent of L2 cache lines at various degrees of sharing

execution and averaged the shared cache line counts of these samples. These shared cache line state counts are grouped based on how many private caches hold cache lines at a time. OpenMP thread counts for each proxy application sweep from 2 to 32 threads in powers of two. We separated the proxy apps by scaling strategy and ordered them by average number of shared cache lines. For most proxy applications, very few cache lines have more than one copy in L2 caches. Only miniVite, CoMD, and XSBench show a significant number of cache lines in shared state. These cache line sharing rates demonstrate how well data is isolated between threads for these proxy applications. In order to correlate this cache line sharing to specific data and sections of code, we examined the program counters of load instructions that resulted in cache lines transitioning to shared state.

MiniVite shows the highest number of shared cache lines for various thread counts, and also shows the highest number of cache lines with high degrees of sharing. MiniVite is a graph analysis proxy app that examines connectivity of nodes in a graph to categorize these nodes into communities. Highly shared cache lines contain nodes in a graph that are connected to nodes in multiple communities. For this proxy application, the number of shared cache lines is high even for low thread counts, and some of the shared cache lines are widely distributed amongst L2 caches. Writes to highly distributed shared cache lines require multiple messages to invalidate copies, increasing the latency of the write operation and increasing traffic in the on-chip network.

We weak scaled the proxy apps for which we had a clear weak scaling option, keeping the amount of data per thread consistent. The proportion of shared cache lines remains similar as threads scaled up for all these weak scaled proxy apps except CoMD. CoMD shows significantly more cache line sharing when it is run with higher thread counts. This trend is an effect of the way we scaled the problem for CoMD. We kept the problem cubic and scaled each dimension by

the cube root of 2 as thread counts doubled. Increasing the X and Y dimensions of this problem increases the surface area between thread data, which explains the increase in cache line sharing between 2 and 8 threads. At 16 threads and beyond, each thread processes less than two layers in the Z dimension. This causes atoms in some boxes to be read by both the previous and the following threads. Because of this, the proportion of shared cache lines greatly increases when the Z dimension is less than twice the number of threads. Maintaining the X and Y dimensions while increasing the Z dimension would control for these effects and eliminate cache line sharing. We confirmed that cache line sharing stayed consistent when we scaled CoMD in only the Z dimension.

XSBench also shows a higher number of shared cache lines as the number of threads increases. This cache line sharing primarily occurs during binary searches of nuclide lookup tables. When multiple threads perform binary look ups of energies from the same nuclide table, they share the first access to the halfway point in the table. These threads share more table accesses depending on how similar their search energies are. Increasing thread count increases the probability of other threads accessing the same parts of the nuclide tables. These nuclide tables are read-only during execution, so we don't expect this cache line sharing to translate to significant inter-cache communication.

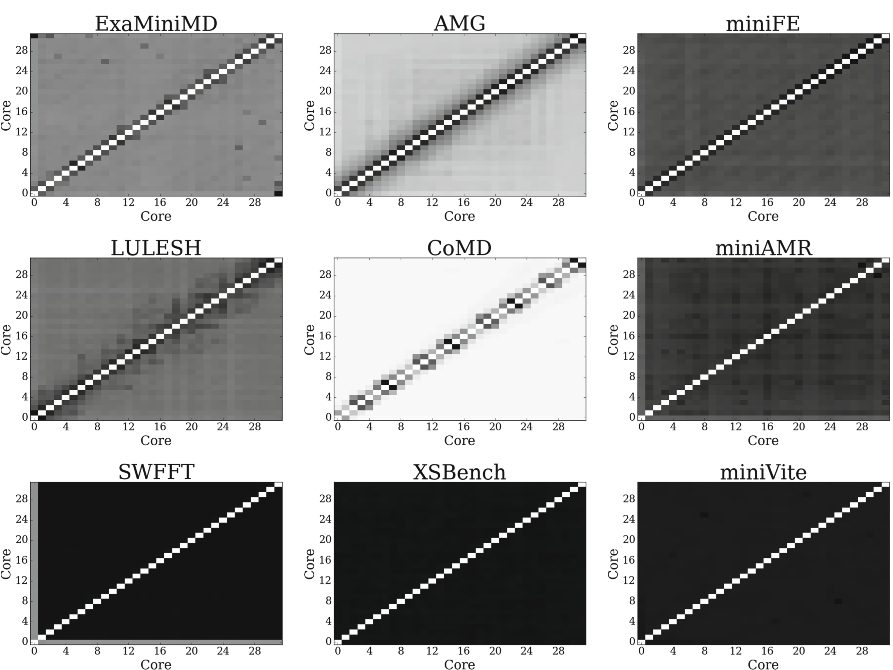

Fig. 2. Cache line sharing pairs with 32 cores

Figure 2 shows the frequency of each pair of cores holding the same cache line in their L2 caches. We collected this data for 32-core simulations. Darker regions of the maps indicate higher cache line sharing. These sharing heat maps offer a visualization of which cores share cache lines and show correlations in data accesses between thread IDs. Shading is normalized to the maximum value for each proxy app in order to show the behavior of these proxy apps, rather than to compare rates of cache line sharing between them. For most of these proxy applications, cache lines are typically shared between consecutive threads. Codes that demonstrate this pattern may benefit from a scheduler that maps consecutive threads to adjacent cores. In the cases of ExaMiniMD, AMG, miniFE, LULESH, and CoMD, each core shares more data with adjacent cores than any other cores. This suggests that mapping logically neighboring threads to adjacent cores will improve data locality and reduce communication delay between common sharers on a chip. This tendency to share with neighboring threads also shows that clustering adjacent cores may be beneficial for these proxy apps. Proxy apps with uniformly shaded maps, such as miniAMR, SWFFT, XSBench, and miniVite, display all-to-all sharing patterns.

While this cache line sharing analysis provides insight into how much data is being accessed by multiple threads, it does not demonstrate how often updates to shared data cause inter-cache communication. An application with a high amount of shared data may never update that shared data, while another application may frequently update relatively few cache lines. We analyze which proxy applications demonstrate frequent cache-to-cache interactions by measuring communication events caused by data updates.

3.2 Producer-Consumer Analysis

In this section, we observe producer-consumer interactions between caches during execution of the proxy applications. These interactions are essentially read-after-write operations. We analyze inter-thread communication from a coherence perspective, showing how often data is still in the producing core's cache when it is consumed. Coherence producer-consumer relationships occur when a consuming thread loads a cache line that exists in a dirty state in another private cache. This coherence communication analysis takes into account temporal access distance and false sharing, showing inter-thread communication that may affect performance. Accesses to remote dirty cache lines cannot be fulfilled by the LLC and require writeback by the private cache of the producing thread. This increases the latency of the consuming request. First, we measure the rate of producer-consumer transactions between caches for each of the proxy apps. We then analyze patterns in these accesses to understand how data moves between threads.

Figure 3 shows the frequency of producer-consumer communication between caches. We display this communication frequency per 1,000 instructions to compare rates between proxy apps and establish an estimate for the frequency of these transactions. Counting events per thousand instructions allows an estimation of the frequency of these occurrences while being agnostic towards the core

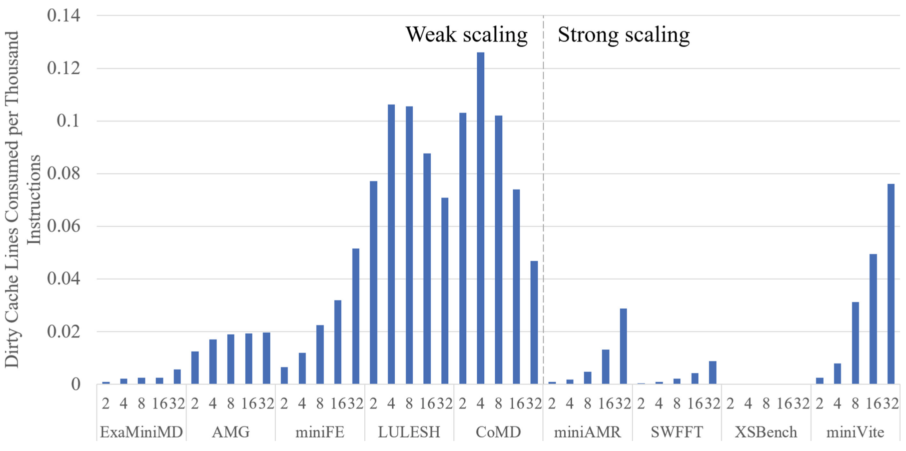

Fig. 3. Consumption rates of modified cache lines

design. Different core designs may execute instructions at different rates and different out-of-order execution capabilities to hide memory latency. Therefore, a simulated core model would be needed to determine the overhead of these communication operations. The normalization per 1,000 instructions is considering instructions executed by all threads, so the rates should be seen as rates per thread as long as the communication happens during parallel sections. If the number of communications grows sub-linear with the number of threads in the weak scaling cases, we would observe a decrease in the number of communication per 1,000 with a linear increase in total instructions. CoMD and LULESH are examples of this behavior, and they exhibit producer-consumer communication more than once per 10,000 instructions for some thread counts. For these two proxy apps, the consuming accesses rate does not increase linearly with thread count past four threads, while instruction counts increase proportionally to thread count due to weak scaling.

MiniAMR, miniFE, and miniVite show consistently increases in communication with higher thread counts. MiniAMR and miniVite are strong scaling cases and therefore see an increase on the total number of communications while the same work spreads across more threads. This trend is unexpected for miniFE, because this proxy app was weak scaled for these experiments. In this case, the number of producer-consumer interactions increases superlinearly with the number of threads.

Figure 4 shows the frequency of each pair of cores exhibiting producer-consumer cache transactions for a 32-core configuration. XSBench has negligible occurrences of producer-consumer relationships between cores with almost no writes to shared data, so its data has been omitted from this figure. To understand the code causing each of these communication patterns and what data is being communicated, we examined the program counters that caused communication between core pairs.

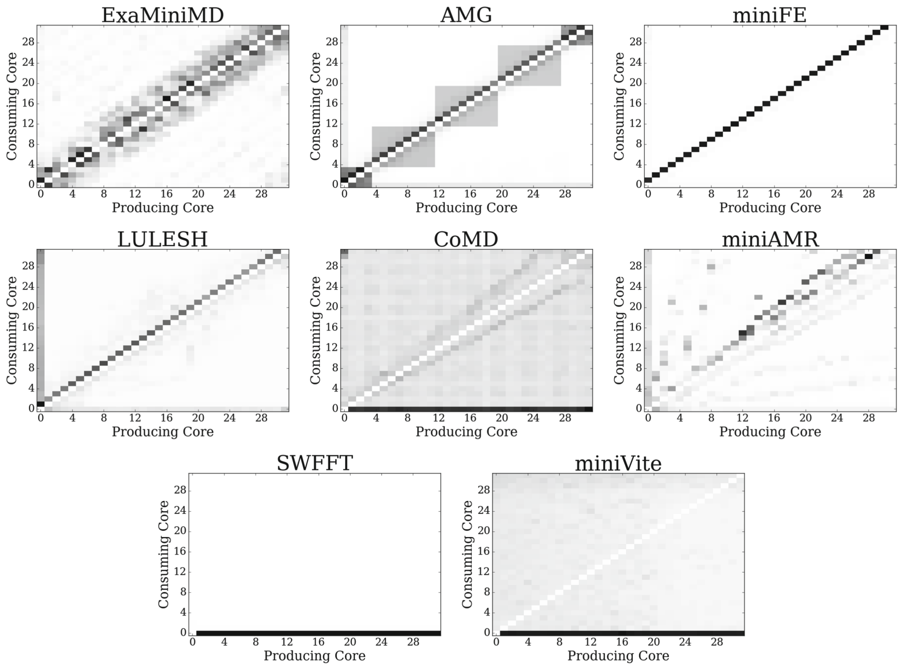

Fig. 4. Producer-Consumer coherence communication patterns with 32 cores

Multiple proxy applications show producer-consumer relationships between neighboring threads. This communication occurs in a single direction for AMG, LULESH, miniAMR, and miniFE, with threads producing data that is consumed primarily by threads with a higher ID. This one-sided communication could occur at the beginning of each iteration after neighboring threads updated their data. The final data updated by one thread would be the first data read by the following thread during a compute interval. Although both the previous and next thread would eventually read the updated data, the data would have been evicted by the producing thread's cache before the previous thread consumes that data at the end of its work iteration. A hardware agnostic evaluation of communication would observe this behavior as symmetric communication, but our coherent analysis reveals this communication may occur between caches asymmetrically and may be predictable, which would enable data to be pushed from producer to consumer in hardware or software.

ExaMiniMD shows symmetrical producer-consumer relationships, where threads produce data that is read by previous or following threads. This may be because ExaMiniMD uses dynamic scheduling, which makes thread interactions less predictable.

The four boxes of all-to-all communication for AMG occur during the *BuildIJLaplacian27pt* routine, when all threads are accumulating into the same array. While each thread accesses a different index of this array for their accu-

mulations, false sharing causes 8 words of this array to map to the same cache line. Therefore, groups of 8 threads perform modifications to the same cache line, causing the cache line to migrate between that group of threads. The smaller boxes of communication between threads 0–3 and threads 28–31 suggest that the beginning of the shared array is offset within a cache line. This false communication could be avoided if the accumulation array is padded such that each thread index in the array maps to a different cache line. The compiler might also be able to avoid this situation by recognizing that threads accumulate to consecutive indices of the array and allocating one cache line for each index. It might also be beneficial if the code was written to utilize OpenMP's reduction capability instead of implementing its own reduction.

Significant communication to or from core 0, such as in the cases of LULESH, CoMD, SWFFT, and miniVite, are caused by serial sections of code. Serial sections may be problematic for scalability, and communication within these serial sections is on the critical path for the entire process, so this might introduce more overhead than communication within parallel sections.

The communication pattern of LULESH shows a one-to-all communication pattern, with significant consumption of data from thread 0 by all other threads. This communication occurs in the libgomp library when thread 0 broadcasts function pointers. LULESH has many consecutive short parallel regions, so this work distribution communication is frequent. This fine-grained parallel loop pattern is detrimental due to the work distribution (fork) and barrier (join) costs, and the serial sections in between loops limiting scaling. A coarser-grained parallelization strategy would mitigate these issues and reduce the amount of one-to-many communications like the ones exhibited by LULESH.

CoMD, SWFFT, and miniVite each show an all-to-one communication pattern, with thread 0 consuming a significant amount of data from all other threads. These consumption patterns occur due to code serialization, where there is a non-parallelized loop with thread 0 iterating over data produced by other threads. For CoMD, serialized reads occur when the atoms in boxes are being updated. This function is serialized per process, which limits scaling with OpenMP threads. For SWFFT, this behavior is the only occurrence of producer-consumer relationships. We observe this behavior when thread 0 distributes data between FFT steps. These serial phases substantially limit OpenMP scalability for SWFFT. For miniVite, serialization happens when thread 0 updates the ownership of graph elements for its process. The serialized loops reduce the parallel efficiency of OpenMP threads due to Amdahl's law. Avoiding serialization or finding a way to parallelize the loops would help scalability.

Preemptively moving data to caches of consuming threads might mitigate some of the overhead of these communications. This could be done in software by cache stashing, or in hardware via data movement prediction or prefetching.

3.3 Write Invalidation Analysis

Write operations to cache lines in shared state experience additional latency, because the write operation must wait for other copies of the cache line to

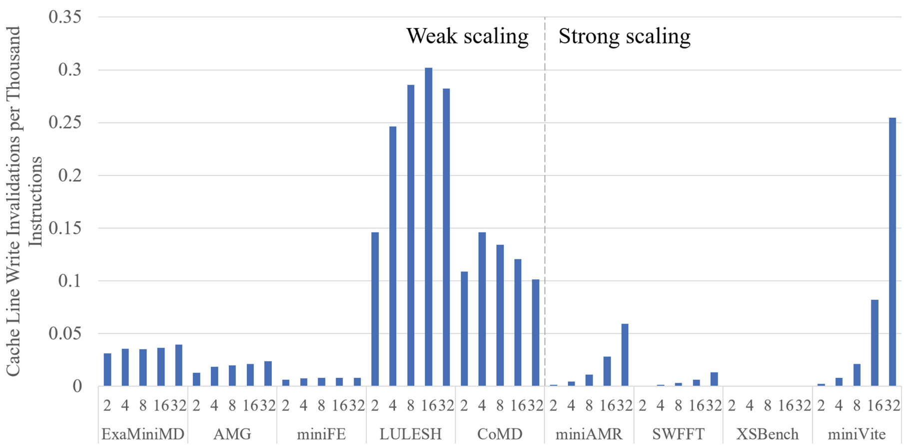

Fig. 5. Write invalidation rates

be invalidated. Write operations to cache lines with more sharers require more invalidation messages, which increases latency and network traffic. The latency might be covered up by an out-of-order core, but the additional network traffic might delay other memory operations. Invalidating cache lines from other caches can also induce future cache misses, which would be unnecessary in the case of false communication. We measured the frequency of write invalidations to understand how this communication occurs in the proxy apps.

Figure 5 shows write invalidations per 1,000 instructions. CoMD and LULESH experience on average at least one write-invalidation every 10k instructions even for low thread counts. MiniFE, miniVite, and SWFFT each show increases in write invalidation rates as thread counts increase.

Figure 6 shows how frequently the core on the x-axis invalidates cache lines from the core on the y-axis. XSBench has negligible occurrences of cache-to-cache write invalidations, so it has been omitted from this analysis. Similar to our previous analysis, we tracked program counters causing these invalidations to find out how this communication corresponds to the code.

For AMG, the invalidations to adjacent threads primarily occur during the relaxation routine. The invalidations between groups of neighbors, which appear as square boxes on the graph, are due to the same false sharing that we observed when analyzing AMG's producer-consumer communication patterns. Padding this array so that threads access disparate cache lines would reduce invalidation traffic as well as producer-consumer coherence traffic.

In CoMD, the invalidated cores are not always adjacent. The writes causing these invalidations occur primarily when threads are sorting atoms in each box after atoms are exchanged. The strange slope of the interactions occurs because the sorting loop is parallelized over the total boxes of the process including halo boxes, while force calculations are parallelized over only the local boxes. Cores 28-31 process halo boxes during this phase, while private caches at this time are

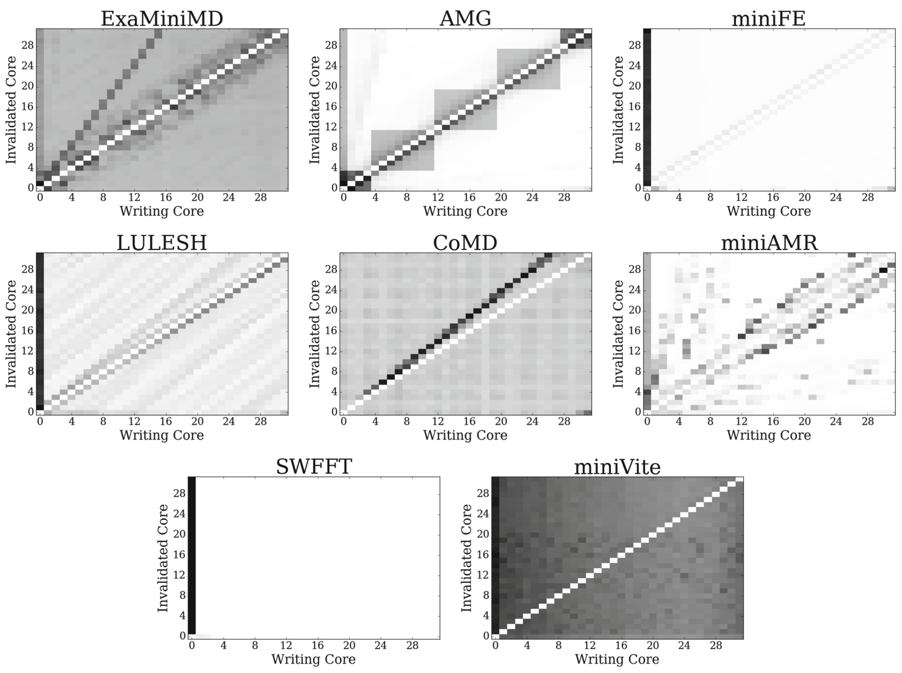

Fig. 6. Coherence invalidation patterns with 32 cores

filled with local box data. This communication may be reduced by splitting the loops that include halo boxes to iterate over local boxes before iterating over the halo boxes. Although this would decrease the write-invalidation traffic, this would introduce additional overhead from adding a separate parallel loop. Cores 30 and 31 invalidate data in core 0's cache because of the serialized updates preceding this operation.

For higher thread counts, miniVite shows a significant increase in write invalidations. Some of these write invalidations occur at the end of the Louvain iteration, when threads are overwriting the communities that nodes belong to. Write invalidations are also observed when information is updated for these communities after the Louvain iteration, overwriting data cached by threads during the iteration. The irregularity of graph accesses results in no discernible relationship between writing cores and invalidated cores.

Write invalidations caused by thread 0 of LULESH primarily occur in libgomp and are caused by the frequent serialization issue that we observed in the producer-consumer analysis.

In SWFFT, write invalidations increase with more threads. All invalidating writes come from core 0, exposing the same issue described in our analysis of producer-consumer patterns. These invalidations occur during the distribution phases, which are not parallelized with OpenMP.

Write invalidations are necessary when caches continue to hold data when it is written to by threads in a different core. This communication could be reduced by flushing cache lines in software when the data is expected to be updated, or the update could be predicted in hardware and flushed from private caches.

4 Related Work

Several publications include characterizations of inter-thread communication for specific multi-threaded workloads.

Barrow-Williams et al. [2] studied communication among threads for the SPLASH-2 [9] and PARSEC [3] benchmarks. Their work observed communication on a word granularity, showing producers and consumers in the application regardless of cache characteristics.

Hillenbrand et al. [6] quantified inter-thread communication for the PARSEC benchmark suite as the number of threads scaled up and measured this communication on a word granularity. Their approach abstracts out the hardware architecture, while our evaluation considers direct cache-to-cache communication.

In contrast to these two works, we examine data consumption and invalidation that occurs between caches at runtime. We include the effects of false sharing and disregard communication operations that do not result in cache-to-cache transactions, i.e., produced data that is evicted before being consumed by another cache. We believe our hardware-focused communication analysis is a better indicator of the impact communication has on performance.

Bienia et al. [3] introduced the PARSEC benchmark suite and characterized scalability as well as cache behavior. The authors measured and reported cache line sharing as the fraction of cache entries in shared state. We account for cache line sharing differently, by counting each unique shared state cache line once. Our approach to measuring cache line sharing shows how much data is shared between caches, without counting copies of the data. Bienia et al. also measured traffic from accesses to shared cache lines, but they did not differentiate by whether these accesses were communicating modified data.

Abadal et al. [1] measured the frequency of multicast operations in a Network-on-Chip during execution of SPLASH-2 and PARSEC benchmarks. They measured these multicasts for broadcast-based coherence as well as a directory-based coherence. The authors state that multicasts in a directory-based design are primarily due to write invalidations, which we measure in this paper.

Richards et al. [8] analyzed the performance of the ECP Proxy Apps with a focus on profiling, instruction mix, cache misses and memory bandwidth. The inter-thread cache-to-cache communication analysis in this paper complements their report with cache line sharing, producer-consumer interactions, and write invalidations in the context of OpenMP thread scaling.

5 Conclusion

In this work, we studied cache line sharing and cache-to-cache communication among OpenMP threads in HPC proxy applications. We identified CoMD,

XSbench, and miniVite as proxy apps that showed high cache line sharing. We then examined how often producer-consumer and write invalidation transactions occur. LULESH, CoMD, and miniVite showed the highest rates of communication among the proxy apps we studied.

Analysis of patterns in coherence traffic between cores running OpenMP threads provided insights into data movement between threads in these proxy applications. This analysis demonstrates to application developers how often communication in their code manifests as cache-to-cache communication at run time. In some cases, the patterns we observe reveal code serialization and false communication between threads. Application developers can use our methodology and results of our analysis to find where to parallelize serial sections of their code that cause considerable data movement, and where they can isolate data used by different threads to prevent false communication. When communication between threads is unavoidable, locality-aware thread placement and improvements in CMP architecture may reduce the overhead of this communication. Our characterization is useful for hardware designers considering data movement optimizations between caches or changes in the coherence protocol.

References

1. Abadal, S., Mestres, A., Martínez, R., Alarcín, E., Cabellos-Aparicio, A., Martínez, R.: Multicast on-chip traffic analysis targeting manycore NoC design. In: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 370–378, March 2015. https://doi.org/10.1109/PDP.2015.26
2. Barrow-Williams, N., Fensch, C., Moore, S.: A communication characterisation of SPLASH-2 and PARSEC. In: 2009 IEEE International Symposium on Workload Characterization (IISWC), pp. 86–97, October 2009. https://doi.org/10.1109/IISWC.2009.5306792
3. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 72–81 (2008). https://doi.org/10.1145/1454115.1454128
4. DynamoRIO. https://www.dynamorio.org/
5. ECP Proxy Apps Suite. https://proxyapps.exascaleproject.org/
6. Hillenbrand, D., Tao, J., Balzer, M.: ALPS: a methodology for application-level communication characterization of Parsec 2.1. In: Proceedings of the International Conference on Computational Science, ICCS 2011. vol. 4, pp. 2086–2095 (2011). https://doi.org/10.1016/j.procs.2011.04.228
7. Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). https://computation.llnl.gov/projects/co-design/lulesh
8. Richards, D., Aziz, O., Cook, J., Finkel, H., et al.: Quantitative performance assessment of proxy apps and parents. Technical report, Lawrence Livermore National Lab (LLNL), Livermore, CA (United States) (2018). https://proxyapps.exascaleproject.org/wp-content/uploads/2018/04/AD-CD-PA-1040PerfCompare.pdf
9. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations. ACM SIGARCH Comput. Archit. News **23**(2), 24–36 (1995). https://doi.org/10.1145/225830.223990