Hybrid MPI/OpenMP Programming on the Tilera Manycore Architecture

Vishwanathan Chandru and Frank Mueller
North Carolina State University, Raleigh, NC

***Abstract*—This work assesses the viability of different programming models for large-scale manycores using an MPI-like abstraction, the vendor's OpenMP, and a combination (hybrid) of both. Experiments with Tilera's TilePro64 demonstrate that MPI and OpenMP both scale while the hybrid model performs inferior to the others. Further, network-on-chip contention significantly affects performance and its variance, especially if the number of utilized cores is high. These findings provide an early insight on trends for single die/chip solutions with large numbers of cores, which will become mainstream in HPC in the coming years. Prior work lacks such a study on large manycores with an efficient native on-chip message passing on shared memory on the same hardware platform.**

I. Introduction

Mainstream HPC systems commonly consist of multiple nodes that communicate via a high-speed interconnect. Two methods of parallelization are generally supported:

1) shared memory parallelization and
2) distributed memory parallelization.

Shared memory parallelism uses OpenMP [1] or the underlying POSIX threads to utilize cores of a node. Parallelism across nodes is coordinated via messages passing, e.g., MPI [2], and synchronization following the SPMD (Single Program, Multiple Data) model of execution. The MPI paradigm can also be used to utilize each core within a node without sharing memory. Both parallelization approaches can also be combined, i.e., leveraging the shared memory infrastructure within a node and high speed inter-node interconnects for MPI across nodes. The challenge for large-scale multicores lies in reducing contention when cores exchange data over NoC interconnects, such as rings and meshes.

The objective of this work is to assess the capabilities and limitations of the MPI, OpenMP, and hybrid programming paradigms for a large manycore with the hypothesis that excessive NoC traffic, e.g., due to MESI-style coherence protocols, may eventually render shared memory ineffective at scale while separate address space for MPI prevent such limitations. We perform this assessment on a Tilera manycore platform [3] with 64 cores on a single die that natively supports shared memory and message passing, which few (if any) other architecture provide at this scale of cores. This is important as manufacturers are switching to mesh NoCs as the core count increase, e.g., for the forthcoming Intel Phi Knights Landing. The assessment is performed using an extension of the NoCMsg library [4], an MPI-like abstraction over the Tilera NoC, which is a prototypical implementation to assess to limits of message passing with nearly the same API and semantics as MPI. This abstraction treats the Tilera manycore architecture as a distributed memory system. It relies solely on low latency software-defined message passing via the NoC instead of architecture-defined shared memory for data communication. Weak scaling [5] and strong scaling [6] experiments are conducted on various benchmarks under multiple configurations. Results show that, depending on the hardware configuration used, some programming model(s) can outperform the others but there is no definitive conclusion that one model is always best, i.e., vendors should support multiple ones. Prior work lacks such a study for up to 56 cores with an efficient native on-chip message passing on shared memory on the same hardware platform (see Section VI).

II. Background

Network-on-chip (NoC) manycore architectures like Tilera [7], Intel Xeon Phi [8] or others [9], [10], [11], [12], [13], [14] provide computing power via large numbers of cores that utilize 2D on-chip interconnects (mesh- or ring-based) instead of traditional bus-based architectures. Efficiency of such architectures is constrained by network contention and memory contention. Tilera's NoC is a switching network where data is split into packets routed across the 2D mesh network with X/Y dimension ordered routing. A core generally interacts with the network via switches using input and output queues. Switches at the intersection of the NoC mesh with multiple input and output queues are then configured to connect cores via a wormhole route for a message request, i.e., links are dynamically reserved along a mesh path, packets are sent, and links are released after sending the last packet.

The Tilera architecture has six mesh networks: (1) IDN (I/O dynamic network) and (2) UDN (User Dynamic Network) are two architecturally-defined networks for routing data among I/O controllers and tiles. (3) MDN (Memory Dynamic Network) is a network for memory data. (4) CDN (Coherence Dynamic network) is used for cache coherence traffic. (5) TDN (Tile Dynamic Network) usage is similar to MDN but it serves requests for tile-to-tile transfers. (6) The STN (static network) is used for data transfer between tiles with fixed routings. Of these networks, only UDN and STN are accessible to users via APIs. The NoCMsg MPI abstraction [4] is based on IPC (Interprocess communication) using UDN, per-core L2 caches, and a global soft L3 cache (composed of multiple L2 caches) that maintains cache coherence via a shared memory protocol with modified/exclusive/shared/invalid (MESI) states.

Tilera implements a non-uniform memory architecture (NUMA) and a non-uniform cache architecture (NUCA). Four memory controllers are placed within the mesh NoC such that cores experience different latencies when accessing memory creating a NUMA abstraction. L2 cache references can further be resolved by a local L2 cache or by the L2 cache of a remote core, which creates a NUCA abstraction. There are three ways to "home" data in the Tilera architecture: Local homing, remote homing, and hash-for-home. In local homing, the entire page (64 kB or 16 MB) is homed on a tile accessing the specific memory and any miss is redirected to the memory controller. This is the default setting for the stack of processes and threads. In remote homing, the entire page is homed on a distant tile. In this case, if the miss occurs at the L2 cache, the data is provided by the remote tile where it is homed. If a data access misses at the remote L2 cache as well, the request goes to memory. In hash-for-home, the entire page is hashed across a set of tiles at cache line granularity.

III. Message Passing Design and Implementation

Tilera supports shared memory natively via a coherence layer and through OpenMP flags during compilation. Hence, this section focuses on the design of an efficient messaging layer, NoCMsg [4], for the Tilera platform. Notice that our design is to applicable mesh-based multicores with native messaging support in general as we have demonstrated by porting our work [15] to the Intel SCC [10].

MPI-like messaging passing over the NoC can be supported by UDN since it has extremely low latency and can be realized at the user level (without operating system overheads). There are two ways to design an MPI-like abstraction using UDN: interrupt based and polling based. iLib [16] is a library that provides basic APIs for data streaming and messaging. Point-to-point communication is supported by iLib and closely resembles MPI semantics but it only supports broadcast and barrier collectives. It is interrupt based and relies on virtual channels and complex packet encoding. OperaMPI [17] is built around iLib using primitives to support more complicated collectives like all-to-all, scatter-gather etc.

Our NoCMsg library [4], on the other hand, is polling based. It is designed around asynchronous work loops to provide flow controlled and non-flow controlled communication. At the core are two work loops:

1) trysend: keeps polling the send queues to check for any pending send;
2) tryrecv: keeps polling and checking on input queues to see if there are any known or unknown receives pending.

These two queues are used for providing flow controlled communication. For non-flow controlled communication, APIs are provided with optional synchronization. These are primarily used to implement collectives.

Fig. 1 and Fig. 2 show the overall design of NoCMsg and OperaMPI, respectively. Due to the polling-based approach, NoCMsg has lower overheads than OperaMPI as the latter utilizes interrupts and more complex protocol messages [4].

The current implementation of NoCMsg supports basic MPI functionality but needed to be augmented for the hybrid model of execution for codes used in the evaluation. We added support for the following features: 1) DOUBLE_INT; 2) MINLOC and MAXLOC reductions; 3) 64 byte unsigned int (uint64); 4) byte datatype; 5) sendreceive functionality; 6) MPI_Get_count; and 7) MPI_Test. Reductions were extended to cover the new data types as well as MINLOC and MAXLOC.

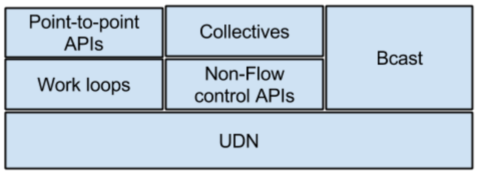

Fig. 1. NoCMsg API design

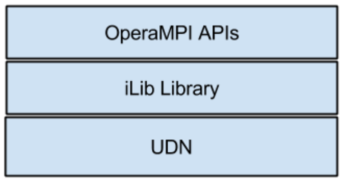

Fig. 2. OperaMPI API design

IV. Evaluation

A. Hardware Platform

We utilized the Tilera TILEPro-64, a single die/chip with 64 tiles/cores, each with a clock speed of 700 MHz with floating point emulation in software. Each core has 16+8 kB private L1 I+D cache(s), a 64 kB private L2 cache and a soft L3 cache of 5 MB, which is created by combining the L2 caches of all tiles. The board has 4 memory controllers, each supporting 2 GB of DDR2 memory. The virtual L3 cache is directory based, where each page is hashed across a specific set of cores. The benchmarks and the NoCMsg library are compiled using the Tilera 3.03 MDE tool chain with the optimization level O3 and the fopenmp flag.

B. Benchmarks and L3 Cache Residency

Two major criteria were considered for selecting a benchmark: 1. Flexibility in terms of process and thread configuration to assess weak and strong scaling for different number of threads; and 2. the per-thread input should fit within the LLC (Last Level Cache) of each tile to bring data via cold misses into the LLC, but no other DRAM accesses will occur from there on. This ensures that we will measure the effect of the NoC (and its contention) on scalability instead of bandwidth limitations on DRAM. We also set the core affinity of OpenMP threads and MPI tasks.

The aim of this work is to analyze the three programming models with respect to the communication overhead. We perform weak scaling in all models (OpenMP, MPI, and hybrid) with inputs per MPI task/thread constrained to the local L2 cache size. This exposes the overheads due to the NoC and minimizes latencies due to memory, which would otherwise dominate and skew the assessment. In the case of weak scaling, the workload is varied proportionately to the number of cores to keep the load per core constant, which ideally should result in constant execution time. Using weak scaling, we can determine how well a problem can scale when increasing the number of cores, which is the objective of this work.

We experiment with 8 to 56 cores. For OpenMP, this translates into 1 MPI task (in case of CoMD) or no MPI task (in case of NAS Multi-zone benchmarks) and varying number of threads. For MPI (NoCMsg) only, we vary the number of MPI tasks and fix the number to threads per MPI task to 1. For the hybrid case (NoCMsg + OpenMP), we use two configurations: 1) We fix the number for MPI tasks to 8 (required to be a power of 2) and vary the number of threads per task as depicted in Fig. 3. This layout excludes the bottom row of 8 cores, some of which are set aside for operating system services by Tilera for a total of 56 user tasks that can be assigned to cores. The layout results in low contention as threads in each circled subset, which represents an MPI task, only share memory with threads within its subset. 2) We fix the number of MPI tasks to 16 and vary the number of threads from 1 to 3 (using rows 0/1, 2/3, and 4/5 of Fig. 3).

To assess the impact of hash-for-home data distribution at L3 level, experiments are conducted with and without hashing. As discussed in Section II, hash-for-home hashes the page at cache line granularity and distributes it across the configured set of tiles/cores in an effort to reduce load imbalance (primarily by using MDN). This has the potential to significantly increase performance due to load distribution and increased LLC cache size, but causes jitter: Even for data that fits into L2 cache, there is a remote access (and thus variable hop count). The resulting contention only gets worse as we increase the number of threads. Even though MDN has twice the bandwidth of UDN [4], we can often get better performance with UDN (NoCMsg only) or with UDN+MDN (hybrid model), as discussed in [4].

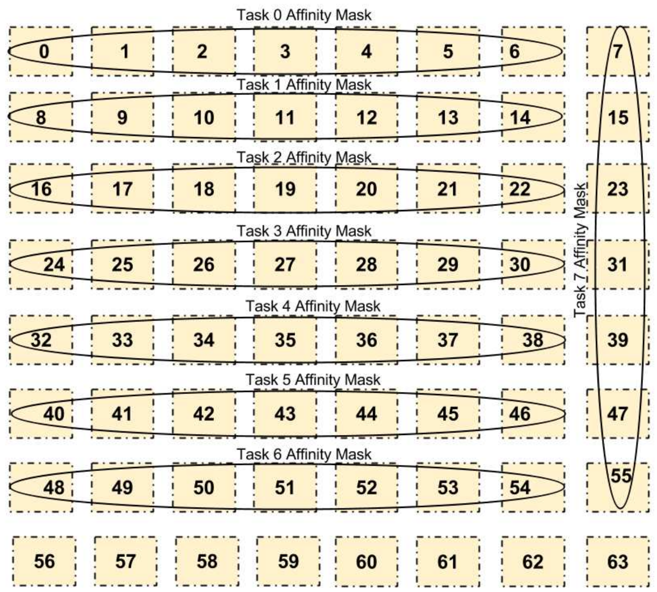

Fig. 3. CPU Affinity used for Hybrid execution (8 MPI tasks)

CoMD is a proxy benchmark for the computations in a typical molecular dynamics application. It is used because it is not constrained in terms of process and thread configuration, and it was possible to reduce the per thread memory footprint to less than 64 kB, which is the LLC cache size per core/tile of the Tilera architecture. It also supports weak scaling. After analyzing the benchmark code and data distribution among threads and MPI tasks, we found that at 108 atoms per thread, the memory footprint is approximately 9.28 kB per thread.

NAS Multi-Zone benchmarks, derived from the original NPB Suite, divide their 3D mesh input into multiple meshes or zones. Each zone is solved independently. After each iteration/time step, boundary values are exchanged between zones. In case of the MPI or OpenMP model, each MPI task/thread is assigned zones but in case of the hybrid model, each process is assigned zones and parallelization within a zone is realized via OpenMP. This approach creates a loose coupling among zones. We utilize the inputs in Figures 4 and 5 for SP-MZ and LU-MZ, respectively, first for weak scaling and then the 49-core configuration for strong scaling. For weak scaling, the number of points per thread remains constant, which ensures a constant computation overhead per core, no matter how many cores are chosen.

# cores	Direction	# Zones	Points per direction	Points per thread
8	x y z	4 4 1	4 4 7	224
16	x y z	4 8 1	4 4 7	224
24	x y z	6 8 1	4 4 7	224
32	x y z	8 8 1	4 4 7	224
40	x y z	8 10 1	4 4 7	224
48	x y z	8 12 1	4 4 7	224
56	x y z	8 14 1	4 4 7	224

Fig. 4. SP-MZ Input Specifications

# cores	Direction	# Zones	Points per direction	Points per thread
8	x y z	4 4 1	5 5 7	350
16	x y z	4 8 1	5 5 7	350
24	x y z	6 8 1	5 5 7	350
32	x y z	8 8 1	5 5 7	350
40	x y z	8 10 1	5 5 7	350
48	x y z	8 12 1	5 5 7	350
56	x y z	8 14 1	5 5 7	350

Fig. 5. LU-MZ Input Specifications

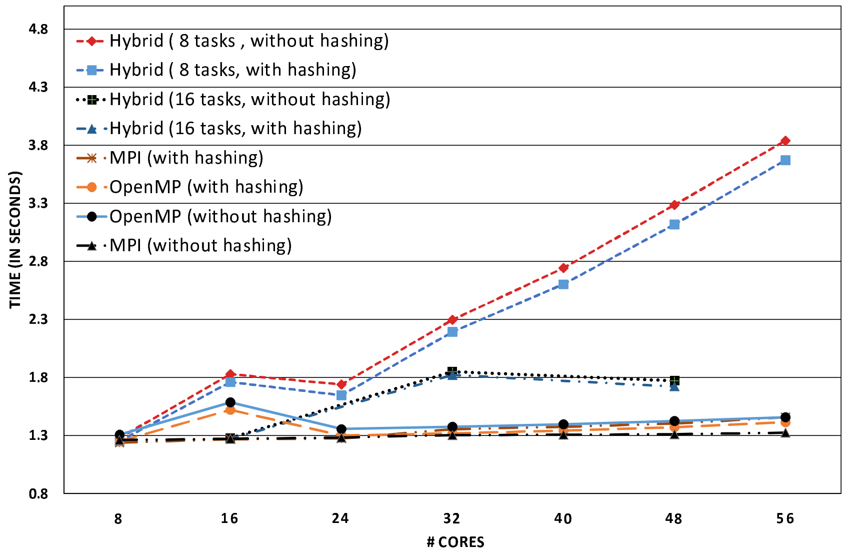

Fig. 6. Runtimes LU-MZ Weak Scaling

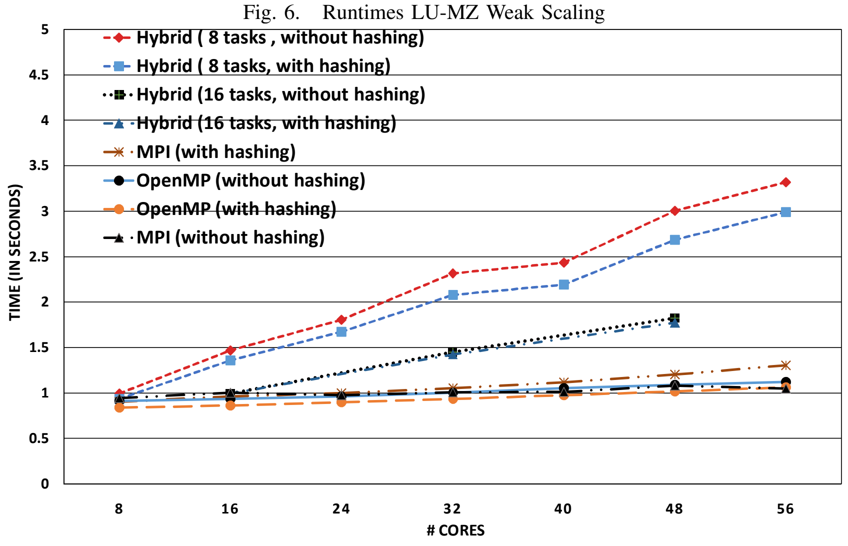

Fig. 7. Runtimes SP-MZ Weak Scaling

V. Experiments And Results

The evaluation assesses the performance of benchmarks using communication via UDN vs. shared memory via the memory interconnects versus a combination of both. We utilize 3 benchmarks, two from the NAS Multi-zone benchmark suite (SP-MZ and LU-MZ) and CoMD. Other NAS-MZ and hybrid codes (CHPMG-FV, Lulesh, CoSP2) do not let us freely vary the number of MPI tasks and threads due to dimensional/algorithmic constraints. All experiments report averages over 15 runs.

Latency due to the NoC plays a significant role in the performance of an application. To analyze the effect of latency, we conduct weak scaling experiments with and without hashing. When we execute the benchmarks without hashing, access is restricted to the local cache, thus minimizing NoC contention and the associated latency. When the pages are hashed across the L2 caches of cores, accesses become remote, increasing the traffic on the NoC and access latencies. Fig. 6, Fig. 7 and Fig. 8 show the average execution times of the three benchmarks in three different programming models, each with hashing turned on and off.

We observe that pure MPI and OpenMP scale well with and without hashing for the NAS MZ benchmarks. Consider LU-MZ (Fig. 6): MPI without hashing consistently performs best followed by OpenMP (where hashing makes little difference) and MPI with hashing. The hybrid configurations do not perform as well (irrespective of hashing). For SP-MZ in Fig. 7, hashing provides a slight advantage for OpenMP while MPI performs best without hashing. Notice that OpenMP outperforms MPI for fewer number of cores but is at par at 56 cores. Again, the hybrid configurations do not perform as well (irrespective of hashing).

CoMD with OpenMP always shows inferior performance compared to MPI and Hybrid irrespective of hashing. This is due the fact that CoMD does not take advantage of the NUMA abstraction in pure OpenMP execution. The operating system policy is to perform an allocation on first touch of a page. Since inputs are initialized before the parallel section, dynamically

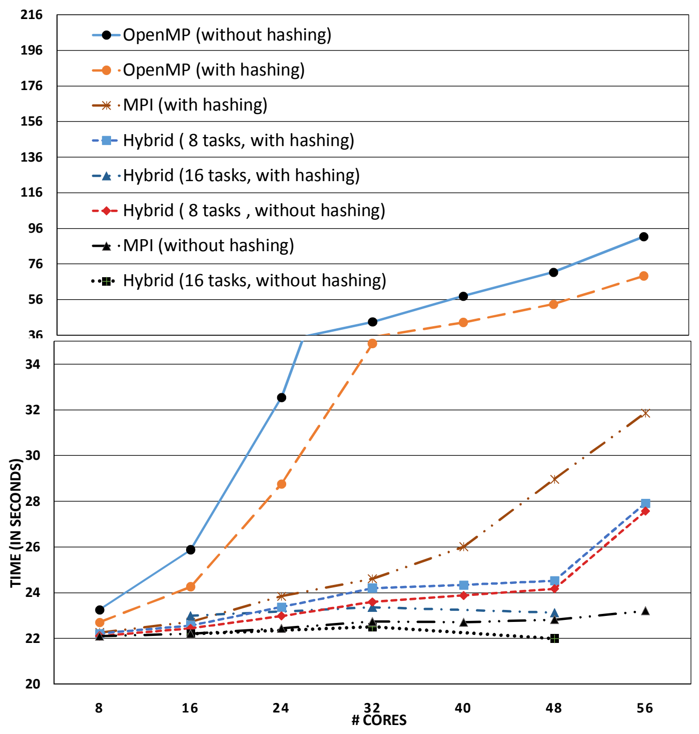

Fig. 8. Runtimes CoMD Weak Scaling

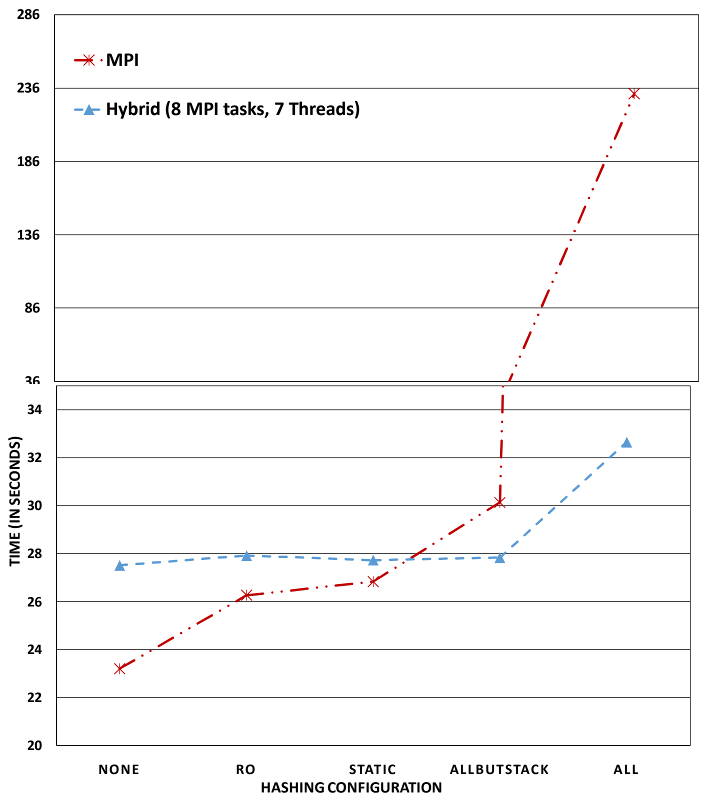

Fig. 9. Runtimes CoMD Hashing 56 cores

allocated private pages are homed local to core 0 (which is where the process is bound to) — instead of the core that is later accessing the page in the parallel section. This leads to high latency and contention due to multiple tiles requesting data from the same home tile and could only be circumvented by refactoring the benchmark source code. For pure MPI, the first touch policy allows places the data at the local core, thus improving performance. Hybrid with more MPI tasks (e.g., 16) also results in better locality to the memory controller for allocations. The NAS benchmarks perform dynamic allocation and initialization after thread creation (in the parallel section), which allocates data locally under first touch. *This shows that it is imperative to initialize data inside the parallel section to obtain good locality under OpenMP on NUMA systems.*

We further observe that CoMD hybrid (with 16 tasks) without hashing outperforms all other methods while other hybrid configurations with fewer tasks (irrespective of hashing) perform worse, as does MPI with hashing. This is due to synchronization overhead, which increases with the number of threads entering/exiting OpenMP sections.

16 MPI tasks hybrid always performs better than the 8 MPI tasks hybrid. In case of CoMD, this is primarily due to a lower threads-per-task ratio, which means fewer threads per home cache, i.e., better locality at the home cache and lower NoC contention. We observe that the runtime closely follows the MPI runtime. From 32 cores (16 MPI tasks and 2 threads per task) to 48 cores, the average execution time goes down slightly compared to the corresponding MPI task configurations. On analyzing the internal timers, we found that the force calculation cost goes up as we increase the number of threads from 1 to 2 but when increasing to 3 threads, the force calculation cost (which mainly utilizes MDN) goes down slightly. An analysis of internal timers of LU-MZ and SP-MZ showed that UDN communication is jittery but the computation cost (directly depending on shared memory) goes down. One reason is that the number of zones per process for 16 MPI tasks is less than that for 8 MPI tasks. Another reason is that the maximum number of threads per process in 16 MPI tasks is 3, which is less than half of that for 8 MPI tasks. As OpenMP parallelization is done per zone, thread synchronization overheads are much higher for 8 MPI tasks. Given the larger number of zones per process in case of 8 MPI tasks, efficiency of parallelization reduces as more threads operate within the same zone compared to zones being assigned to threads for pure OpenMP. This potentially results in loss of locality, which further increases the NoC contention. Thus, we observe better performance for 16 MPI tasks.

When we enable hashing, the performance dynamics change due to increased NoC contention. Hashing can be configured in five different ways [18]:

1) *none*: nothing is hashed;
2) *ro*: only the text section and read-only data is hashed;
3) *static*: text, data and bss sections are hashed;
4) *allbutstack*: except stack, which is locally cached, everything is hashed;
5) *all*: everything is hashed.

We utilize *allbutstack* for our comparative experiments. As we increase the number of cores used, the additional NoC

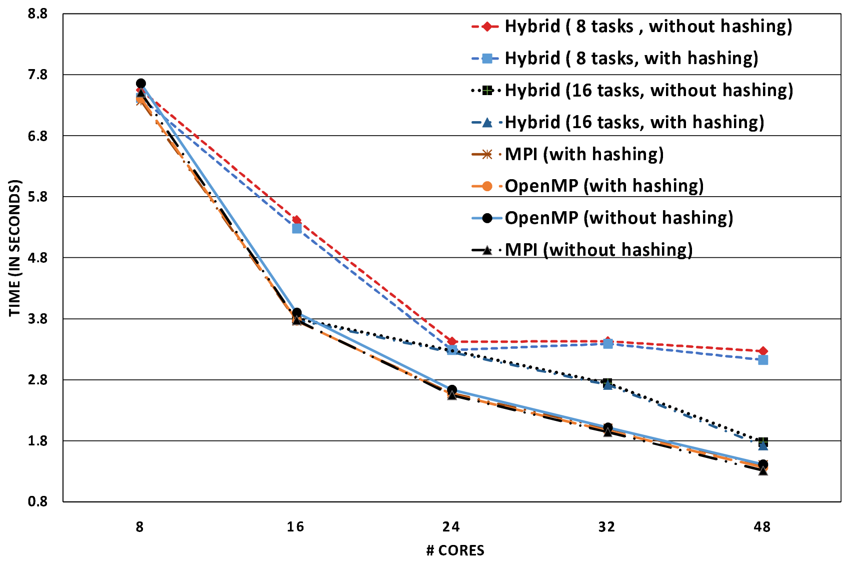

Fig. 10. Runtimes LU-MZ Strong Scaling

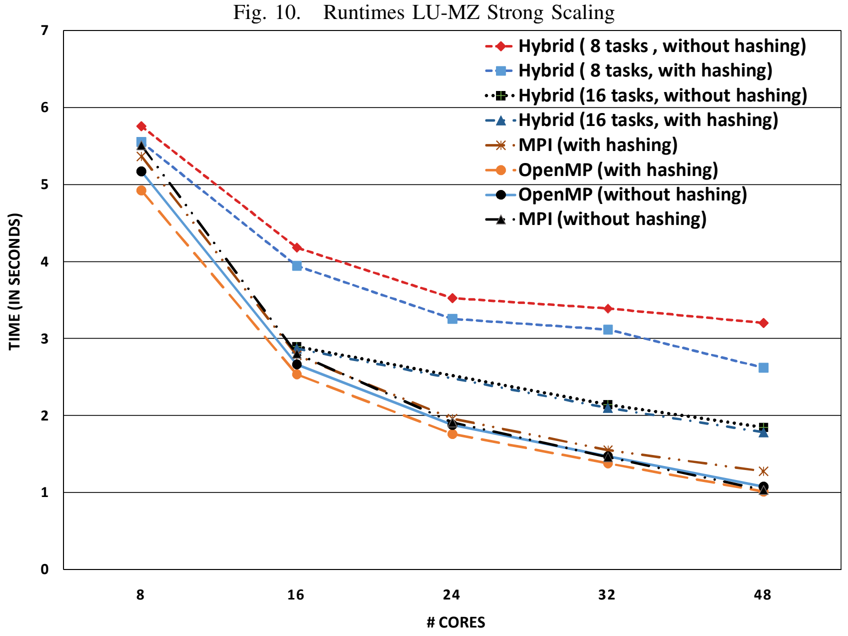

Fig. 11. Runtimes SP-MZ Strong Scaling

contention results in a slight performance deterioration for MPI and Hybrid-8 for both LU Fig. 6 and SP Fig. 7. OpenMP still benefits from hashing, even with larger number of cores. For CoMD, hashing improves the performance of pure OpenMP, but OpenMP in general does not perform well due to lack of allocation locality, as discussed. The MPI version of CoMD in Fig. 8 suffers significantly under hashing as the number of cores increases due to NoC contention.

Due to NoC contention, the variance in execution time without hashing is higher than with hashing and increases as we increase the number of cores. Generally, the configurations performing worse also result in much higher variance in execution time across all benchmarks.

We designed an experiment to understand the impact of different types of hashing. We ran on 56 cores with pure MPI and hybrid varying the types of hashing. Fig. 9 shows CoMD results (others omitted but are similar). From *allbutstack* to *all*, we observe a significant decline in performance for pure MPI and a slight one for hybrid, even though CoMD allocates its key data structures dynamically. This shows that stack hashing is the reason for the poor performance of pure MPI compared to hybrid for hashing. (Notice that the upper half of the results is on a coarser scale than the lower, which allows us to fit the results but causes a "step" at 34seconds).

We also conducted strong scaling experiments. For LU-MZ and SP-MZ for the 48 core input (i.e., 96 zones and number of points per zone as per Fig. 5 and 4, respectively). For CoMD, the strong scaling input is fixed at 4896 atoms per test configuration. The input specifications of strong scaling exceed the L2 cache capacity for all core counts less than 48, but when hashing is enabled the input fits within the soft L3 cache (created by combining the L2 caches of all tiles).

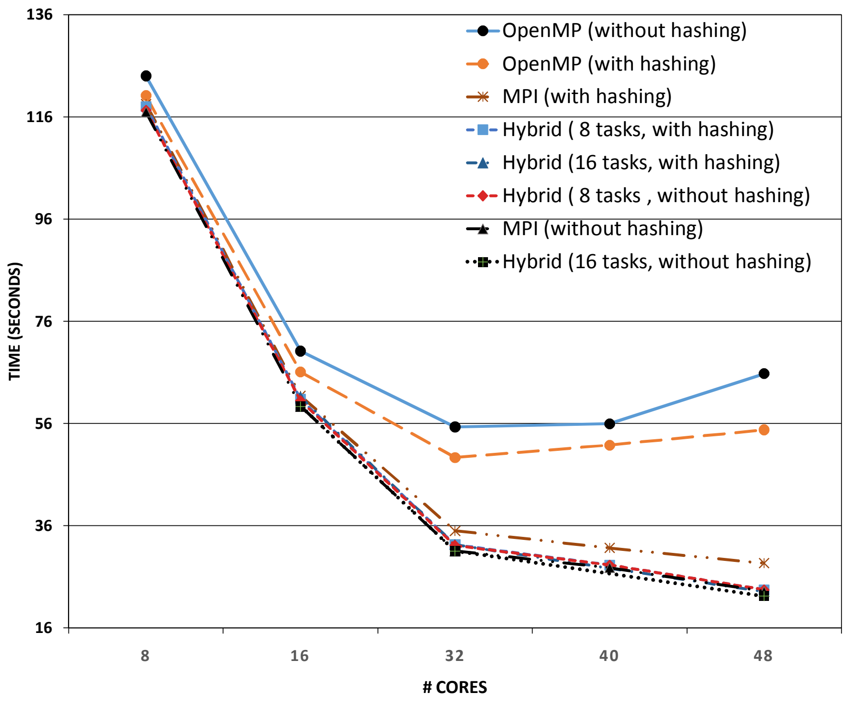

Fig. 12. Runtimes CoMD Strong Scaling

Fig. 10, 11, and 12 show the average runtimes of LU-MZ, SP-MZ and CoMD over various processor counts, respectively. (Some configurations are omitted due to input constraints.)

The average execution time is on par with the expected decrease in runtime as we increase parallelism (OpenMP, MPI or Hybrid). But in some cases the performance starts to deteriorate due to increased NoC contention, which counters the potential improvement, e.g., for CoMD under OpenMP in Fig. 12. For up to 32 cores, performance improves for all configurations, but after that performance deteriorates for OpenMP while the other configurations continue to experience performance benefits, albeit at a lower slope after 16 and then again 32 cores.

We also observed that jitter (SD) is high for hashed executions compared to non-hashed execution for CoMD but less so for NAS MZ benchmarks. This is due to NoC contention resulting from non-local allocations for the former while NAS MZ has smaller memory footprints per task, which mediate the ashing effect.

Contention for strong and weak scaling arises due to different reasons. In case of weak scaling, the input specifications are designed such that the data fits in the LLC. With hashing turned off, the likely source of contention is UDN for pure MPI, UDN+Cache coherence networks (MDN, TDN and CDN) for hybrid, and cache coherence networks for pure OpenMP. With hashing turned on, the dynamics change and the source of contention becomes the cache coherence network due to randomized data placement in remote L2s.

In case of strong scaling, the inputs are designed to fit into the soft L3 cache but they exceed the individual L2 size of a core. With hashing turned off, contention is due to off-chip memory accesses (MDN) along with UDN (in case of pure MPI) and cache coherence networks in case of the hybrid model. For OpenMP, contention is mostly due to the cache coherence networks. With hashing turned on, the complete input fits the soft L3 but almost all the accesses become remote (served by L2 caches of distant cores), causing more contention on cache coherence networks compared to MDN and UDN alone.

VI. Related Work

Hung et al. [19] analyze the performance gain and parallelization on many-core platforms for object detection, which is a computationally intensive problem. They propose a performance prediction equation for object detection, which is then verified via experiments. Serres et al. [20] assess the performance and scalability of UPC (Unified Parallel C) over GASNet based on Pthreads and OperaMPI on the Tile64 many-core architecture. They analyze Pthreads and MPI separately but not hybrid OpenMP+MPI. Suh et al. [21] present a performance analysis of FFT and CRBLASTER on the Maestro processor (derived from the Tilera architecture), while Singh et al. [22] analyze the performance and scalability of FFTW (Pthreads-based parallelization) and CAF (iLib-based shared memory parallelization) but neither MPI nor hybrid. Martin et al. [23] provide an analysis of cache coherence and shared caches within the context of scalability via simulation but do not consider the latencies introduced due to a NoC. Jost et al. [24] and Rabenseifner et al. [25] compare various programming models on a cluster with SMP nodes and find benefits for the hybrid mode. In contrast, we study scalability on manycores and the NoCMsg lower-cost MPI on-chip abstraction, which leads to no advantage of hybrid over MPI. Zimmer et al. [4] focus on comparing NoCMsg and OperaMPI to find that the former significantly outperforms the latter, which is why our study focuses on NoCMsg. They only briefly evaluate OpenMP but do not assess hybrids while we cover hybrids and perform more detailed scalability studies with CoMD and the latest NAS MZ hybrid codes. Joven et al. [26] evaluate MPEG decoding implemented as hybrid MPI+OpenMP on a 32-core ARM manycore, which is only cache coherent up to 8 cores. In contrast, Tilera is coherent up to 64 cores, which allows us to assess the limits of OpenMP in comparison with MPI and the hybrid model. Furthermore, we focus on NoC contention, which is a novel contribution.

VII. Conclusion

We show that programming models affect performance and scalability and are subject NoC latencies. Pure MPI is highly scalable for a large manycore with a mesh NoC. But when we induce hashing, we see a significant deterioration in performance for larger numbers of cores due to stack sharing. OpenMP follows closely while hybrid tends to trail behind except for CoMD, in part due to its serial memory allocation. We observe that data initialization *inside* the parallel section is required for good locality and performance under OpenMP on NUMA systems. However, if such parallelized initialization is not feasible or cannot provide locality for all application phases due to alternating access patterns, pure MPI and hybrid without hashing perform best whereas OpenMP and any scheme with hashing create NUMA bottlenecks.

We conclude that NoC latencies and contention are a key consideration for scalability on many-core architectures. Tilera offers a number of optimization opportunities due to six NoCs, each dedicated for a certain purpose. Such a choice can benefit performance at scale as the developer may select the programming model that provides the best fit for an algorithm and its data layout/partitioning. Our recommendation is for vendors to provide such choices in their software stack.

Acknowledgment

Tilera Corporation provided technical support of the research. This work was funded in part by NSF grants 1239246 and 1058779 as well as a grant from AFOSR via Securboration.

References

[1] *Official OpenMP Specification*, www.openmp.org, May 2005. [Online]. Available: http://www.openmp.org/drupal/mp-documents/spec25.pdf

[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sep. 1996.

[3] "Tilera processor family," www.tilera.com.

[4] C. Zimmer and F. Mueller, "Nocmsg: Scalable noc-based message passing," in *International Symposium on Cluster, Cloud and Grid Computing*, 2014.

[5] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[6] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.

[7] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, pp. 15–31, 2007.

[8] "Intel xeon phi," https://www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html, Apr. 2015.

[9] "Adapteva processor family," www.adapteva.com/products/silicon-devices/e16g301/.

[10] "Single-chip cloud computer," blogs.intel.com/research/2009/12/sccloudcomp.php.

[11] "SCC External Architecture Specification (EAS) Revision 0.94."

[12] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference*. ACM, 2007, pp. 746–749.

[13] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom *et al.*, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*. IEEE, 2010, pp. 108–109.

[14] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Lger, B. Orgogozo, J. Reybert, and T. Strudel, "A distributed run-time environment for the kalray mppa-256 integrated manycore processor," *Procedia Computer Science*, vol. 18, no. 0, pp. 1654 – 1663, 2013, 2013 International Conference on Computational Science.

[15] K. Yagna, O. Patil, and F. Mueller, "Efficient and predictable group communication for manycore nocs," in *International Supercomputing Conference*, Jun. 2016.

[16] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE micro*, no. 5, pp. 15–31, 2007.

[17] M. Kang, E. Park, M. Cho, J. Suh, D. Kang, and S. P. Crago, "Mpi performance analysis and optimization on tile64/maestro," in *Proceedings of Workshop on Multi-core Processors for SpaceOpportunities and Challenges Held in conjunction with SMC-IT*, 2009, pp. 19–23.

[18] *Programming The Tile Processor*, Tilera, "www.tilera.com".

[19] Y.-F. Hung, S.-Y. Tseng, C.-T. King, H.-Y. Liu, and S.-C. Huang, "Parallel implementation and performance prediction of object detection in videos on the tilera many-core systems," in *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*. IEEE, 2009, pp. 563–567.

[20] O. Serres, A. Anbar, S. Merchant, and T. El-Ghazawi, "Experiences with upc on tile-64 processor," in *Aerospace Conference, 2011 IEEE*. IEEE, 2011, pp. 1–9.

[21] J. Suh, K. J. Mighell, D.-I. Kang, and S. P. Crago, "Implementation of fft and crblaster on the maestro processor," in *Aerospace Conference, 2012 IEEE*. IEEE, 2012, pp. 1–6.

[22] K. Singh, J. P. Walters, J. Hestness, J. Suh, C. M. Rogers, and S. P. Crago, "Fftw and complex ambiguity function performance on the maestro processor," in *Aerospace Conference, 2011 IEEE*. IEEE, 2011, pp. 1–8.

[23] M. M. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Communications of the ACM*, vol. 55, no. 7, pp. 78–89, 2012.

[24] G. Jost, H. Jin, D. an Mey, and F. F. Hatay, "Comparing the openmp, mpi, and hybrid programming paradigms on an smp cluster," in *Proceedings of EWOMP*, vol. 3, 2003, p. 2003.

[25] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*. IEEE, 2009, pp. 427–436.

[26] J. Joven, A. Marongiu, F. Angiolini, L. Benini, and G. D. Micheli, "An integrated, programming model-driven framework for noc-qos support in cluster-based embedded many-cores." *Parallel Computing*, vol. 39, no. 10, pp. 549–566, 2013.