ABSTRACT

CHANDRU, VISHWANATHAN. Analysis of Memory Performance and Execution Models for Large-Scale Manycores. (Under the direction of Dr. Frank Mueller.)

Platforms consisting of many computing cores have become mainstream in scientific computing/high performance computing, but present a technological challenge to others like real-time embedded systems. While they provide increased processing power and system availability, they can impose latencies and contention for memory accesses as multiple cores try to reference data at the same time. This becomes critical for hard real-time systems where predictability and performance isolation are important. On a multiprocessor board with 50 or more processing cores, the NoC (Network on Chip) adds to that unpredictability. This work makes two contributions:

- It develops a bank-aware and controller-aware allocator and it evaluates the impact of targeted allocation on performance isolation and contention. Experiments for bank-aware and controller-aware allocation show that such targeted allocation significantly improves performance isolation, results in less contention and reduces worst case and average execution times.

- It utilizes an MPI-like abstraction designed over the NoC and the vendor's OpenMP support for the assessment various programming models. The experiments are primarily designed to assess the impact of NoC contention and load distribution on application performance. The results demonstrate that MPI and OpenMP abstractions are most scalable given the right data decomposition but they are also highly susceptible to NoC contention. The hybrid model, which utilizes a mix of both paradigms, performs inferior to MPI and OpenMP in most of the cases but is most resilient to NoC contention. Results further indicate that NoC contention can significantly affect the variance of performance, especially if the number of utilized cores is high. We further conclude that compared to small multicores, optimization on a large manycore platform is governed by a different parameters, such that one model sometimes performs better than others depending on the configuration.

Analysis of Memory Performance and Execution Models for Large-Scale Manycores

by
Vishwanathan Chandru

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2015

DEDICATION

To my parents.

BIOGRAPHY

The author was born and brought up in Indore, a city popular for street food in Madhya Pradesh, India. He pursued his Bachelor's degree in Information Technology at Vellore Institute of Technology University at Vellore, India. After the undergraduate degree, he worked at GE Healthcare on Medical Diagnostic Imaging software for 3 years till 2013. Then he began his Masters at NC State University from Fall 2013 in the field of Computer Science. He started working as a Research Assistant in Systems Research Group under Dr. Frank Mueller in Summer 2014.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Frank Mueller for his help, guidance, patience and encouragement. I would also like to thank Dr. Xipeng Shen and Dr. Guoliang Jin for agreeing to be in my thesis committee and helping me through the process.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Mainstream HPC systems commonly consist of multiple nodes, where each node features one or more multi-core CPU. A node commonly features a NUMA architecture, a high speed data bus, multi-level cache hierarchies and other advanced features along with optional co-processors like GPUs (Graphical Processing Units) to speed up computation and communication. These nodes communicate via a high-speed interconnect. Two methods of parallelization are generally supported:

- shared memory parallelization and
- distributed memory parallelization.

Shared memory parallelism, MIMD (Multiple Instruction Multiple Data and SIMD (Single instruction multiple data), rely on sharing memory or distributed shared memory for data exchanges. This approach predominantly uses threads for parallelization. OpenCL, CUDA and OpenACC for GPU-based parallelization, OpenMP or Pthreads (in a Linux environment) for CPU based parallelization are examples of this category. But this kind of parallelization cannot take advantage of multiple nodes in a HPC cluster. Distributed memory parallelization generally follows the SPMD (Single Program, Multiple Data) model of separate processes, which allows multiple nodes within a cluster to be utilized. It requires explicit communication over the high-speed interconnect, subject to latencies of the memory hierarchy within each node. The solution to the limitations of both parallelization approaches is to use them in combination, i.e., leveraging the shared memory infrastructure within a node and high speed inter-node interconnects to perform processing across nodes. This is also known as the hybrid MPI/OpenMP programming model. The goal of all three programming models is to minimize the communication delay and to maximize computation. Therefore, if we want to maximize the performance of an application, inter- and intra-MPI task data distribution is critical.

Determining data distribution (and therefore the work distribution) is the key for an efficient parallel application, but it is not trivial to achieve. Often, depending on the topology of the interconnect, the communication pattern, memory footprint, intra-node bandwidth, inter-node bandwidth and load distribution, a coarse domain decomposition at the process level and a coarse or fine sub-domain decomposition within a process are required. One also needs to consider the characteristics of individual nodes, e.g., NUMA hierarchy, cache design, number of cores and sockets, etc. For example, if we consider the pure MPI model for a communication-bound application and treat each core within a node as a CPU, processes running within a node have a higher communication bandwidth than the inter-node bandwidth, taking advantage of shared memory within the node. If an application is parallelized in a topology-aware manner, we can obtain higher bandwidth and lower latencies resulting in superior performance. The hybrid model of parallelization can lead to reduced communication and superior performance if domain and sub-domain decomposition are performed properly. For example, if we consider a CPU-bound application, we can use a co-processor, like a GPU or Intel Xeon-Phi, to speed up the computation using the hybrid programming model.

With multicore/manycore platforms with up to 100's of cores and proposals to scale the number of cores even further combined with an evolution from bus-based architectures to ring/mesh based Network on Chip architectures [Tili],[Ada],[Sccb],[Scca],[Bor07],[Wen07b], manycores resemble a distributed system with optional shared memory. The challenge becomes to determine the best partition scope and size for OpenMP sub-tiles, and to identify techniques that limit shared memory contention. In such architectures, a hybrid programming model can be beneficial. The key aspects are to reduce communication overhead, to improve load balance and to reduce computational overhead.

On many-core platform(s), memory (DRAM) is a resource critical to performance. As applications share cores and become more and more memory intensive, DRAM tends to become a performance bottleneck that critically affects system performance [WM95].

Performance problems usually arise due to serialization of memory accesses. This can be avoided using bank-aware and controller-aware allocation. The DRAM in many-core platform(s) is divided among multiple memory controllers and, within a controller, ranks and banks. Controllers and banks can be accessed in parallel. Therefore, performance of an application varies significantly depending on how data is placed and how many cores/processors access a given bank at same time. In the best case, each core/processor accesses a different controller/bank. This ensures that contention of accesses does not occur and that accesses are resolved in parallel. One strategy to improve bank-level parallelism is to use bank interleaving. In case of single threaded applications it improves performance. But if multiple programs or multiple threads are running simultaneously, it can cause cross-interference. The higher the degree of parallelism, the higher is the probability of bank sharing resulting in more cross-interference.

Linux on the Tilera platform can be configured to allocate memory on the closest NUMA node (physical memory controller). But since Linux on the Tilera platform handles DRAM as a single resource following a NUMA allocation policy (unless and until disabled), banks are not considered during allocation and it is not possible to predict the exact location of allocated memory over the banks [Tild; Tila].

Multiple works both at the software and hardware level are being pursued to improve predictability and performance isolation. Some approaches like Paolieri et al. [Pao09] and Akesson et al. [Ake07] focus on scheduling of requests to improve performance and predictability. Another technique improves performance via assigning cores private banks [Wu13] [Rei11]. All of them require hardware modifications. There are software-based approaches like PALLOC [Yun14] and Liu et al. [Liu12], which propose modifications at the OS level to implement bank coloring. But when it comes to manycore architectures like Tilera, contention due to the NoC also becomes critical.

Unlike a bus connecting cores to other cores and components, the Tilera architecture features a mesh NoC instead of a bus. The *Intel Xeon Phi co-processor* [Mic] is another example of a many-core architecture co-processor that utilizes an *Inter-processor Network ring* instead of a bus. All data accesses and data exchanges go through the NoC. Given the large number of cores, traffic over the network can lead to high latencies. Therefore, NoC contention also becomes very important for predictability and performance isolation. Due to the large number of cores, controller contention also becomes a critical factor.

Figures 1.1 and 1.2 show the best-case and worst-case scenarios considered in this work. The worst case does not arise due to bank-level contention but due to NoC contention and a specific memory access pattern. If all accesses are directed via a particular memory controller as shown in Fig. 1.2, then the latencies increase since requests get queued at controller(s) and core(s) stall. Furthermore, cores in each quadrant may access all controllers, which leads to a high volume of traffic. Even though all controllers are being utilized, the latency is high.

1.1 Hypothesis

We hypothesize that a mesh-based network-on-chip (NoC) interconnects of a modern manycore architecture benefit in performance from

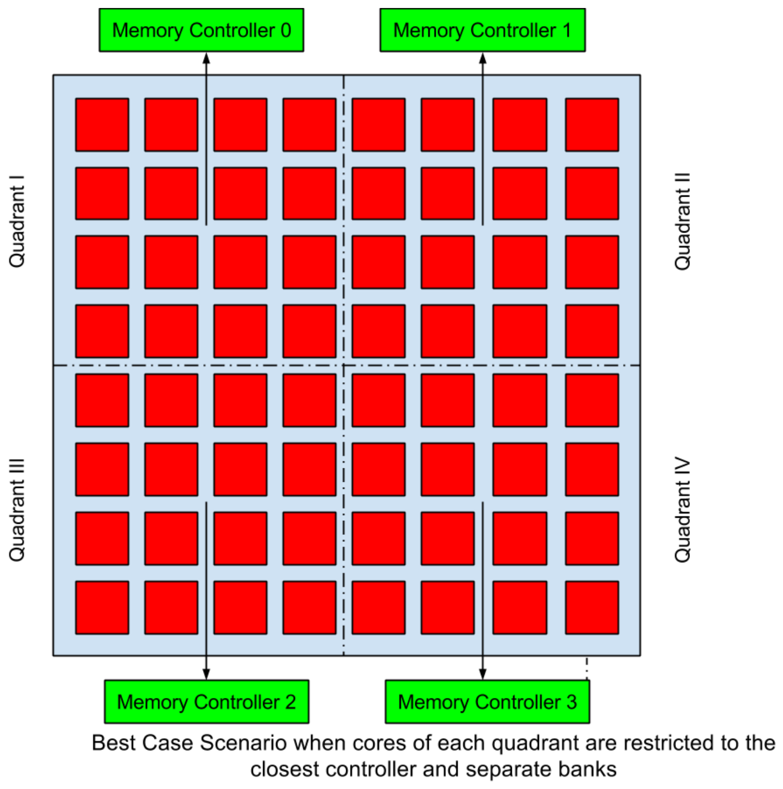


Figure 1.1 Best Case Scenario

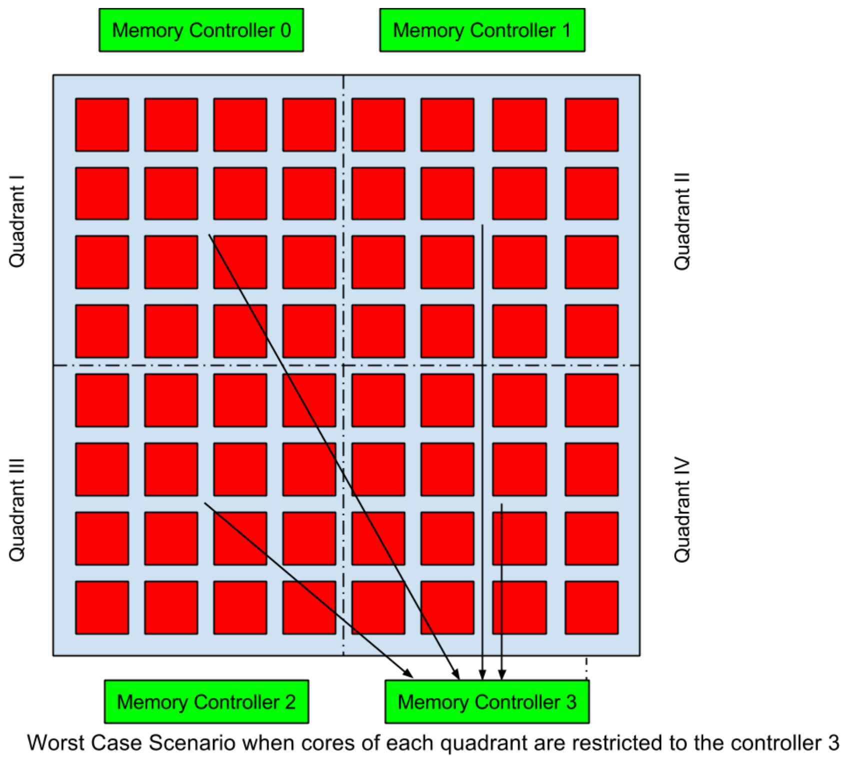


Figure 1.2 Worst Case Scenario

- controller- and bank-aware memory allocation and
- a hybrid MPI+OpenMP programming model due to reduced NoC contention and more balance and localized utilization of shared resources

1.2 Contributions

The work makes the following main contributions:

- It introduces a user space bank-aware and controller-aware allocator that keeps track of bank(s) and controller(s) of allocated memory, i.e., it returns memory addresses requested on a particular bank/controller. This allows users to bind a core/processor to a specific bank and controller reducing the contention and serialization, thus improving performance isolation. Using the allocator, an extensive experimental study was performed to evaluate the impact of bank-aware and controller-aware allocation on performance isolation. Various benchmarks, including STREAM, NPB's IS (OpenMP version) along with a composite benchmark generated using the Mälardalen benchmark suite, were used.
- It assesses the hybrid programming model in comparison to pure shared and pure distributed memory parallelization on the Tilera architecture [Tili] . The assessment is performed using an extension of the NoCMsg library [ZM14], an MPI-like abstraction over the Tilera NoC. 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 [Gus88] and strong scaling [Amd67] is performed on various benchmarks under multiple configurations.

Chapter 2

Targeted Allocation

2.1 Background

The Tilera [Tili] architecture differs from most modern systems due to presence of a mesh NoC instead of a bus. Other platforms with large core count now use a mesh or ring NoC as well (Intel Xeon Phi [Mic], Intel SCC [Sccb], Adapteva [Ada] etc.). Like modern DRAM systems, the memory system is composed of a controller that handles the arbitration, scheduling and conversion of packets to external memory commands. Memory is organized into ranks (4 in our setup) and each rank has multiple banks (8 in our case), which can be accessed in parallel. Each bank is composed of a row buffer and an array of storage cells organized into rows and columns. The systems may or may not have multiple independent memory channels. The Tilera architecture supports four channels capable of independent operation [Tile]. Data is accessed by issuing an ACT command to get data into a row-buffer. Then, the column can be accessed. If another row has to be accessed, the data in the row buffer has to be written back with a PRE command before activating another row. Since DRAM storage is composed of capacitors, a periodic refresh is necessary. Due to timing constraints and operational methodology we can observe:

- a high access latency for closed rows exists.
- operations on different banks may proceed in parallel.

Tiles (similar to cores) do not have direct access to controllers as the architecture implements a DSM (Distributed Shared Memory) protocol to ensure coherence at the L2 level cache. Tilera utilizes a *mesh interconnect* for data exchange [Tilf]. Dimension ordered routing is used in the network [Tile]. The interface to off-chip memory and I/O devices is done via *I/O shims*.

They are the interface between memory controllers and other I/O devices [Tilf]. The memory controller has multiple MDN ports [Tile] where requests arrive and are fulfilled. To optimize memory bandwidth and latency, reordering is performed based on the following considerations:

- the overhead of the memory accesses and DRAM page policy on memory side;
- load balancing and starvation avoidance on the tile side.

Due to these architectural considerations, the NUMA allocation policy is the default. If, however, the STRIPE mode [Tile; Tild; Tilc] is enabled, a page (64 kB size) is striped across all controllers in an interleaved way at 8 kB granularity to balance load and improve memory parallelism. Another strategy to load balance is hash homing [Tilb], where a page is striped at cache line granularity and distributed among the available tiles. However, NUMA and STRIPE allocations might not be able to ensure performance isolation as the accesses to the data structure will be resolved by different memory controllers with different latencies (hops) over the NoC.

First, none of the policies are bank aware, i.e., they cannot restrict accesses of a task running on a tile to a particular bank or even controller if memory striping is enabled. And even with a controller-aware NUMA policy, contention may not necessarily be reduced: Allocation will always be performed on the closest controller, but if all tasks are placed close by, contention may actually increase.

The second reason is bank sharing. Even if memory is striped across the controller(s) to improve load balancing and throughput, the probability of bank sharing is still high. Thus, bank contention may decrease throughput for memory intensive tasks. NoC contention is another overhead in case of memory striping. Since a memory page is effectively distributed across controllers, the probability of NoC contention increases significantly due to longer NoC data paths when we increase the number of tasks and the size of the memory footprint.

To limit the impact of bank sharing, bank address hashing [Tile] is performed at the hardware level to randomize the banks within a page based on the physical address of a page. This increases the bank availability but that becomes less significant as we increase the memory footprint of the tasks and the number of tasks.

In the worst case, both issues simultaneously manifest. In this paper, we present a user space allocator utilizing non-striped mode and using controller-interleaved page allocation instead of the default the NUMA policy. We use the interleaved policy for allocating our memory pool as it ensures that an equal amount of memory per controller is available for allocation.

2.2 Design

In this paper, we propose the design and implementation of a user space bank-aware and controller-aware allocator. It exploits the virtual to physical address translation. After determining the bank and controller from bits within the physical address, the address is added to a specific list corresponding to the relevant controller and bank. When a user requests memory from a certain bank and controller, the corresponding list is searched and if memory is available, its address is returned.

Our allocator requires pre-allocation of a large pool of memory, which is then traversed at a granularity of 8 kB and chunks are added to the corresponding free list(s). The default behavior of the allocator is to try to find a memory chunk fitting the requested size, bank and controller. But if the exact request cannot be full-filled, the allocator supports multiple modes as a fallback from the default behavior, namely:

- *CONTROLLER RESTRICTED*: If a requested bank is not found within the controller, the allocator defaults to the bank that has the most free memory.
- *CONTROLLER UNRESTRICTED*: If a requested bank and the controller cannot be used to satisfy the request, the allocator defaults to the bank and controller that have the most free memory.
- *SPLIT*: If the requested size is greater than 8 kB and this mode is enabled, then standard allocation is performed (i.e., find the first fitting contiguous chunk available) and its address is returned.

Algorithm 1 shows the implementation. Multiple modes can be configured via a bit-mask for finer control of the allocator. Multiple lists are used to improve the performance of the allocator (see Fig. 2.1). Multiple doubly linked list(s) are maintained, one for each bank within a controller. The head of the list is indexed using the controller number/bank number. For SPLIT allocation, a separate list is maintained for quicker allocation. Each free memory chunk is part of both lists. Each memory chunk has four pointers, two pointers that are used to traverse the corresponding bank list and two pointers for the list used for SPLIT allocation.

As we can see from the allocation modes and the algorithm, we perform book-keeping on memory available per bank/controller. For this purpose, a queue is maintained in which elements are sorted according to memory availability and its doubly linked list implementation, i.e., both forward and reverse traversal is possible. Five such queues are maintained, one queue per controller to keep track of banks and one queue to keep track of memory available per

```
Input: Request Size, Memory Controller, Bank, modes mask
block ⟵ NULL;
block ⟵ Memory from requested controller and bank;
if block != NULL then
    return block;
end
if CONTROLLER_RESTRICTED in modes_mask then
    Q ⟵ Banks corresponding to requested Memory Controller with banks in
    descending order of memory availability;
    for each bank in Q do
        block ⟵ allocate memory from current bank if chunk of size greater than or
        equal to requested size is available;
        if block != NULL then
            return block;
        end
    end
end
if CONTROLLER_UNRESTRICTED in modes_mask then
    ControllerQ ⟵ Get controllers in descending order of memory availability;
    for each controller in ControllerQ do
        BankQ ⟵ Get banks from current controller (ordered in descending order of
        memory availability);
        for each bank in BankQ do
            block ⟵ allocate memory from current bank if chunk of size greater than or
            equal to requested size is available;
            if block != NULL then
                return block;
            end
        end
    end
end
block ⟵ NULL;
if SPLIT_MODE in modes_mask then
    block ⟵ Allocate contiguous block of memory (may span across banks and
    controllers);
end
return block;
```

Algorithm 1: Bank aware algorithm allocator

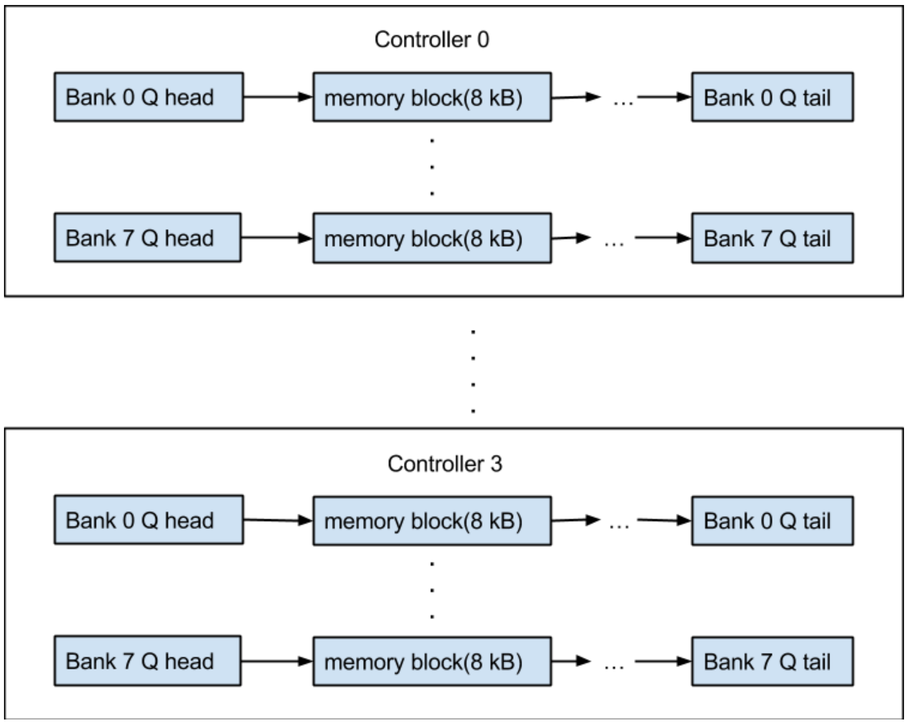


Figure 2.1 Queue(s) used in allocator

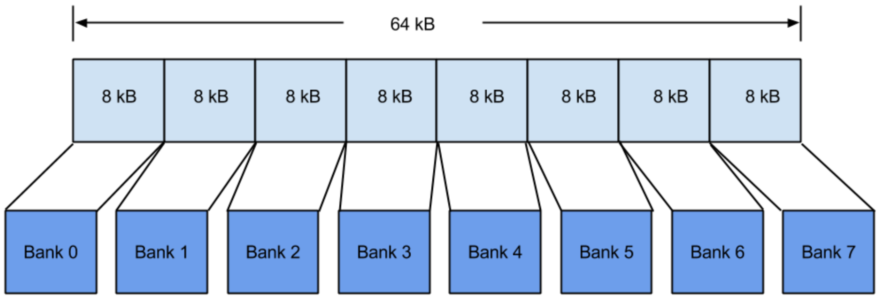


Figure 2.2 Bank split up in a 64 kB page(with no address hashing)

controller. Queues are array based for fast operation as the number of banks and controllers is fixed.

Even though multiple levels of design abstractions are introduced, they primarily accelerate bank-aware allocation (or any allocation less than 8 kB). SPLIT allocation will be slow as multiple data structures and free lists need to be updated. For this reason, all our experiments exclude the allocation time and focus on real-time applications after they pre-allocate data during the initialization.

One shortcoming of our approach is that the allocator works best for shared memory tasks. For non-shared memory tasks, it will lead to wasted memory as a separate pool will be needed for each task.

In contrast to PALLOC [Yun14], a kernel-level allocator, and other software partitioning approaches [Jeo12] [Liu12], our allocator works in user space. It is more restrictive in terms of usage due to multiple reasons. Due to Tilera's design of a 64 kB page size and utilization of controller-interleaved page placement, we cannot allocate more than 64 kB of contiguous memory within a controller. As the bank varies every 13 bits (explained in the address mapping section), a contiguous allocation within a bank cannot be more than 8 kB (see Fig. 2.2). Also, due to allocator design restriction(s), the allocator cannot perform contiguous allocation within a controller greater than 8 kB (see Fig. 2.1). Unlike PALLOC, this allocator is not seamlessly integrated, i.e., explicit modifications are required to utilize this allocator in accordance with the previously mentioned restrictions.

2.3 Evaluation

2.3.1 Hardware Platform

The evaluation platform used is a Tilera TILEPro-64 with 64 tiles [Tilg]. Each tile has 16+8 kB private L1I+D cache(s), a 64 kB private L2 cache and a soft L3 cache of 5 MB, which is created by combining the private L2 caches of all tiles. There are 4 memory controllers, each capable of independent operation. Each controller can support up to 32 banks (4 ranks and 8 banks per rank). The configuration we utilize has 8 GB of DDR2 RAM, 2 GB per controller. Address hashing [Tile] is enabled to enhance the number of available banks.

Memory striping is disabled in the test configuration to obtain a more predictable worst case execution time. When memory striping is enabled, Linux sees only one controller and a page is striped across all four controllers. To determine which 8 kB chunk belongs to which physical controller, we would need to write an additional wrapper function over Tilera's API. In contrast to this, if memory striping is disabled, all four physical memory controllers are visible to Linux, i.e., just by appropriately setting the affinity of a task and by using the default NUMA policy, we can restrict allocations to a controller. This also makes the simulation of the pathological worst-case scenarios easier as we can explicitly set the affinity of a tile (core) to a memory controller using Tilera's API.

2.3.2 Address Mapping

Address translation is straight forward. The physical address has 36 bits [Tile]. Per the documentation and configuration register values, address hashing is performed to increase bank availability. Bits 13, 14, 15 are used to determine banks and bits 34 and 35 are used to determine the controller.

The implication of the position of bank-determining bits is that a bank changes every 8 kB. Since the Tilera architecture dictates pages to be 64 kB in size, one page is spread across 8 banks. The configuration was determined by reading from a SPR (Special Purpose Register) [Tile] using the Tilera Board Test Kit [Tilh].

We also need to consider bank address hashing while determining the banks. Algorithm 2 shows the bank determination functions.

```
if hash[0] = 0 then
    bank[bankbits-1:0] = addr[bankbits+colbits+2:colbits+3];
end
if bankbits+colbits+2 >= 15 and bankbits=3 then
    high[bankbits-1:0] = 1'b0, addr[17:16];
end
if hash[0] = 1 then
    bank[bankbits-1:0] = addr[bankbits+colbits+2:colbits+3] $\wedge$ high[bankbits-1:0];
end
```

Algorithm 2: Bank Address Hashing

After querying the SPR controller configuration, we determined that colbits is 10 and bank bits is 3. *addr* in Algorithm 2 is the physical address. As we can see from the functions described in Algorithm 2, we take bits 16 and 17 from the physical address and XOR them with the identified bank bits. This results in a randomization of banks based on the physical address. Based on the previously mentioned function, we assume that the first 8 kB of a page is bank 0. However, after hashing, it can be any bank from 0-3.

Apart from the default configuration of 8 banks, we also experimented with 2 and 4 bank configurations. As the bank index ranges from 0-7, to restrict ourselves to two banks, we wrap around the bank index after every second bank. Similarly for 4 banks, we wrap around the index

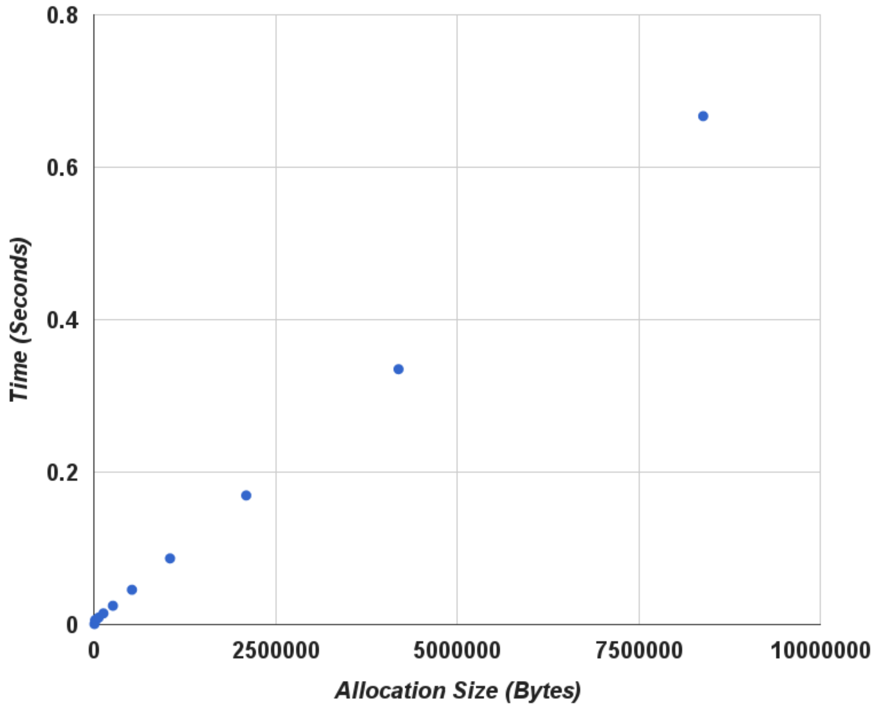


Figure 2.3 Allocator performance in SPLIT mode as we increase the request size

every 4th bank, i.e., after banks 0-3. All this is done during bank identification for building the free list so no configuration changes on the platform are required.

2.3.3 Overhead

As discussed in the previous section, the allocator is fast for requests of size less than or equal to 8 kB. But if we allocate contiguous memory greater than 8 kB, we need to use the SPLIT mode, where the performance varies depending on the size of allocation. Fig. 2.3 shows the timing plot for varying allocations in SPLIT mode. As we increase the size of allocations the difference between allocations keeps increasing. An allocation of 8 MB requires close to 0.7 seconds but an allocation of an 8 kB chunk takes only about 0.007 seconds. As real-time systems tend to allocate memory ahead of time, a detailed analysis on allocator performance under various conditions is omitted.

2.3.4 Key Ideas

The experimentation mainly focuses on *same bank allocation vs. varied bank allocations*. Same bank allocation is not possible due to bank address hashing. So even when we access the first 8 kB of each page, it is randomized to any bank between 0-3 (not necessarily 0). Thus, when we try to perform same bank allocation via Tilera's API and default allocation policy, there is some bank parallelism. One way to avoid this is to write a wrapper around Tilera's API to identify the chunk belonging to bank 0 and return it for accesses. But we did not implement this as we want to see the effectiveness of bank address hashing in reducing bank contention.

Second, consider data dependencies (RAW, WAW and WAR). These dependencies lead to a stall as re-ordering cannot take place. If we can define the access pattern that violates temporal and spatial locality, we will always encounter a cache miss and one outstanding memory request at a time. On Tilera, this can be achieved by disabling caching at all levels while performing allocations using Tilera's API. To compile, the O3 flag is used for all experiments to optimize the code.

To measure bank-level contention, we use OpenMP benchmarks and a composite benchmark. The entire processor is divided into 4 quadrants with 16 tiles each, except quadrants 3 and 4 with only 12 tiles each as the bottom-most 8 tiles cannot be used for execution in our setup. The primary aim of dividing tiles into quadrants is to restrict the access to memory via the closest memory controller. This allows us to remove controller contention and to minimize NoC contention so that we can focus on measuring bank level contention. This also helps if we want to create pathological worst case scenarios, i.e., tiles from each quadrant accessing all the controllers to generate excessive memory contention in all quadrants accessing controller 0 and 1. The execution time is indicative of contention. Colored allocation results in lower execution time compared to non-colored allocation, and we get better memory bandwidth with colored allocation.

During the entire experiment, threads were pinned to the respective tiles to prevent them from migrating and causing unintended interference.

2.3.5 Benchmarks

A multi-threaded composite benchmark was designed. Each thread is assigned one kernel and the number of times threads execute the associated kernels can be specified (also termed as iterations). Once launched, threads execute each iteration in a lock-step manner. This is ensured using two barriers. Minimum, maximum and average execution times are reported per thread along with the standard deviation. We have modified and integrated bubble sort, nth largest and

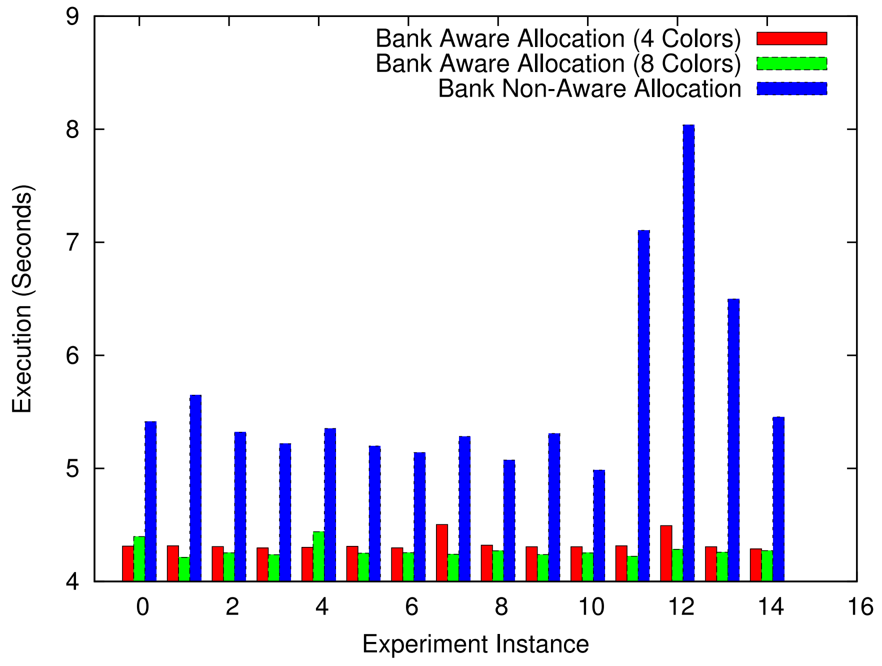


Figure 2.4 Modified NPB IS OMP (Class A) Execution

stats from the Mälardalen Benchmark Suite [Gus10]. The modifications are limited to changing key arrays from 1D to 2D.

We have also utilized the NAS IS OMP [Bai91] and STREAM [McC95] benchmark(s) for our experiments. The modifications in these codes are also limited to changing key arrays from 1D to 2D and then performing dynamic allocation either using our allocator or using the Tilera's API. OpenMP parallel sections, if present, are modified to set affinity of individual threads as well.

2.4 Results

Let us assess the impact of bank-aware allocation on performance. We use execution time to measure the worst-case performance impacted by latency, bandwidth and contention. By running the same benchmark in multiple tile(s) and using an OpenMP enabled benchmark, the impact of bank-aware and controller-aware allocation is shown. The *x-axis* of the plots denotes the i^{th} experiment/execution (experiment instance) and the *y-axis* shows memory bandwidth (MB per second)/execution time (seconds) depending on the plot.

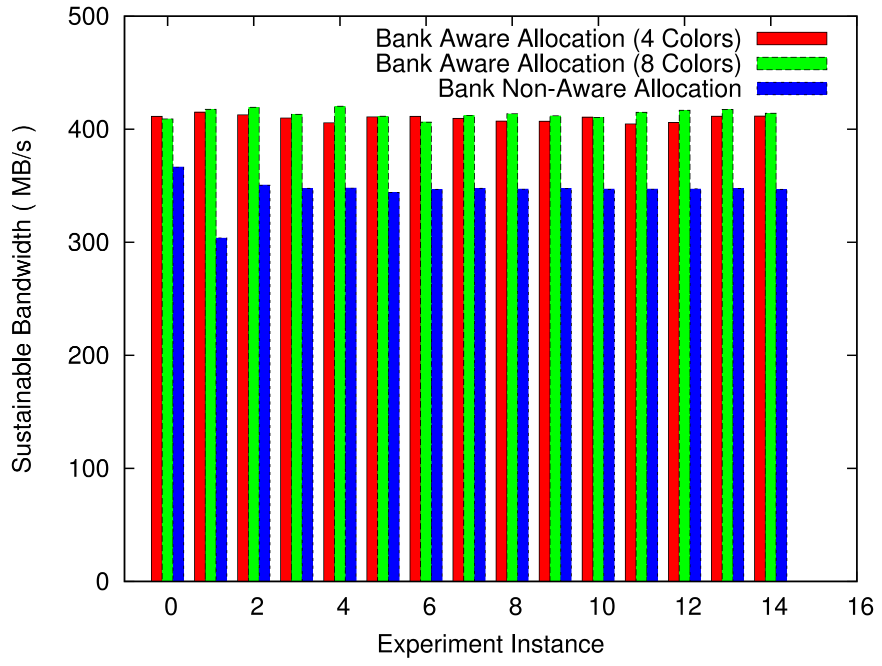


Figure 2.5 Modified STREAM Add Sustainable Bandwidth

Fig. 2.4 depicts the execution time of the NAS IS OMP benchmark for 32 threads (8 threads per quadrant). We obtain close to a 20% performance benefit with colored allocation. Fig. 2.5 (bandwidth reported by modified STREAM benchmark with 32 threads and 8 threads per quadrant) shows a significant increase in sustainable bandwidth for the ADD kernel from the modified STREAM benchmark.

To further understand NoC contention and bank contention, composite benchmark executions (for the modified bubble-sort from the Mälardalen benchmark suite) with varied number of threads and memory footprints were performed. Bubble-sort is used as it is the most memory bound of all integrated benchmarks. As mentioned in the benchmark section, composite benchmark runs are reported in terms of iterations. To get reliable data and account for outliers, multiple iterations are performed per execution and multiple rounds of executions are run per experiment. Fig. 8 shows the plot of overall maximum execution time per experiment of the composite benchmark for 32 threads and 2 pages per thread. We observe that colored allocation always results in the best performance isolation. The two pathological worst case scenarios, tasks in each quadrant accessing all quadrants (shown as circular allocation in the plot) or tasks restricted to a particular controller always lead to the worst performance. This highlights the

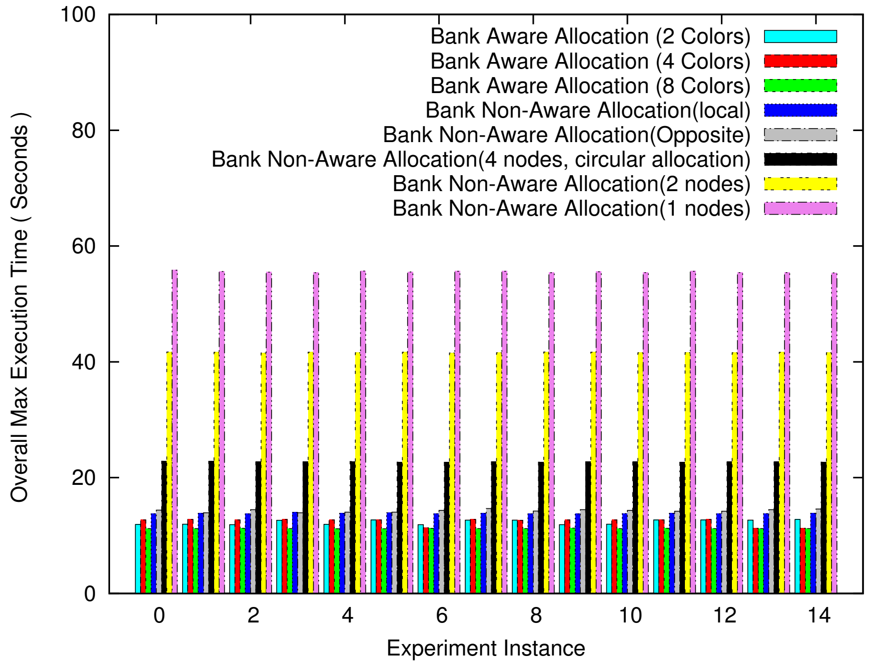


Figure 2.6 Composite Benchmark (32 Threads, 2 Pages per Thread)

criticality of NoC contention and controller contention. Controller-local allocation is slightly slower than bank-aware allocation within a controller (colored allocation) as the latter reduces serializations of accesses.

The difference between 2, 4 and 8 color allocations is noticeable. This is due the fact that even though we are simulating two and four bank configuration(s) by wrapping around bank indices when populating free lists at user level, at hardware level we still have 8 banks, which is a higher bank level parallelism than expected for a four or two color scheme.

To further analyze the impact of colored allocation, the following two sets of experiments were performed:

1. Constant memory footprint and variations in the number of threads.
2. Constant number of threads and variable memory footprint per thread.

Fig. 2.7 shows the overall maximum execution time per experiment/execution of the composite benchmark for 32 threads and 10 pages per thread. As we compare the plots in Fig. 2.6 with Fig. 2.7, we observe a drastic increase in execution time mainly due to increased contention and due to the large number of elements to be sorted. An improvement of 17% to

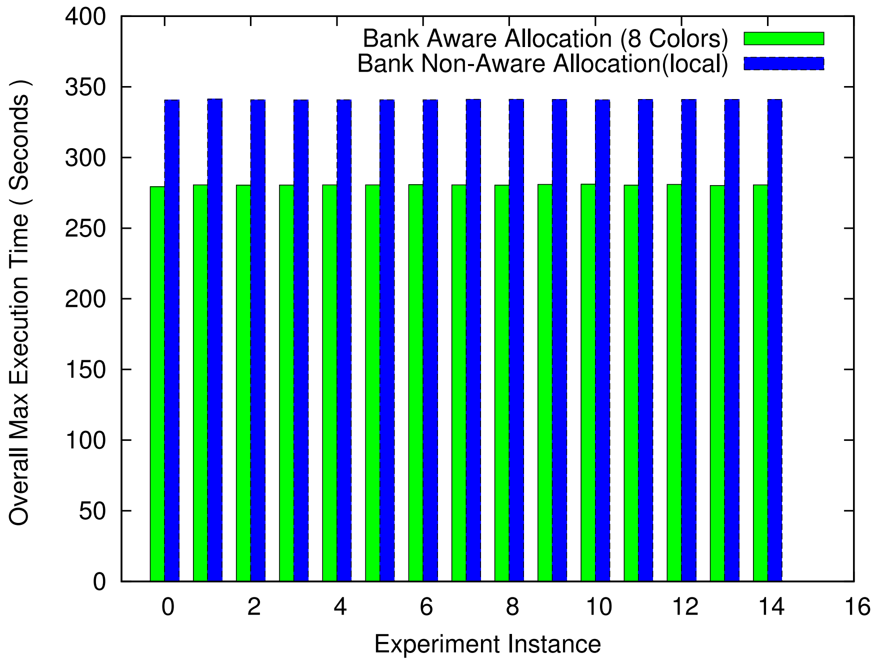


Figure 2.7 Composite Benchmark (32 Threads, 10 Pages per Thread)

20% over non-bank aware controller-local allocation was observed on average. This difference increases from less than 3 seconds in case of 2 pages per thread to 60 seconds in case of 10 pages per thread. The demand for bandwidth increases due to a larger memory footprint. But since bank-level parallelism increases available bandwidth by reducing the serialization of accesses and cross-interference, we observe that as threads become more memory bound, the impact on performance of bank-aware allocation increases.

For experimental case 1, we fixed the memory footprint to 10 pages per thread and varied the number of threads from 8 to 32 in steps of 8, i.e., increments of 2 threads per quadrant. We observe that the difference between colored allocation and bank-unaware controller-local allocation keeps increasing until 32 threads, and then remains close to 60 seconds (see Fig. 2.6, Fig. 2.7 and Fig. 2.8). This is due to the fact that each controller has 8 banks per rank, i.e., for up to 32 threads we were able to restrict each thread in the quadrant to a separate bank. But as we exceed 8 threads per quadrant, more than 1 thread accesses a given bank leading to increased contention. Now, as we reduce the number of threads to 8, non-colored controller-local allocation provides good performance most of the times (see Fig. 2.9). There are two possible reasons for this behavior.

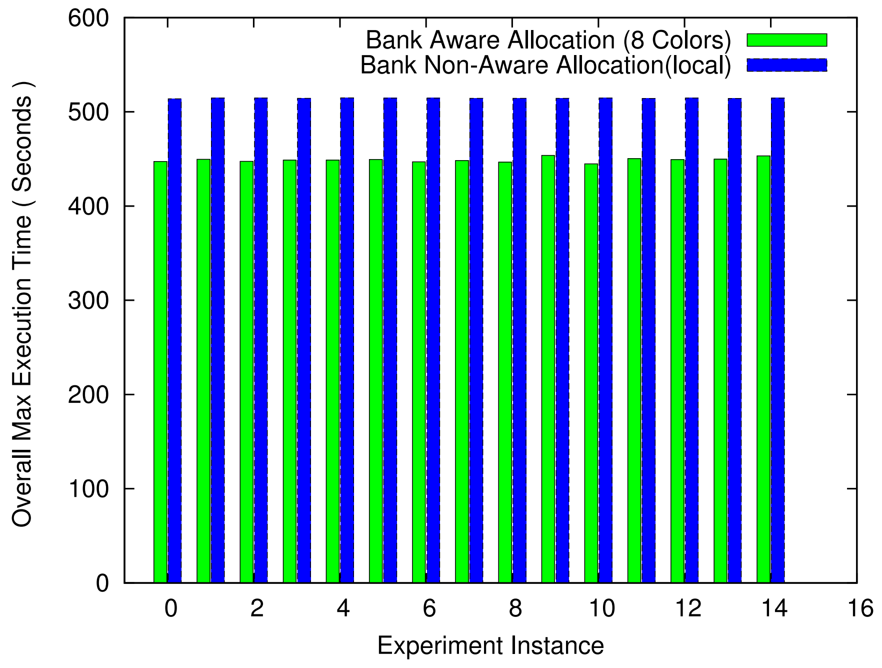


Figure 2.8 Composite Benchmark (48 Threads, 10 Pages per Thread)

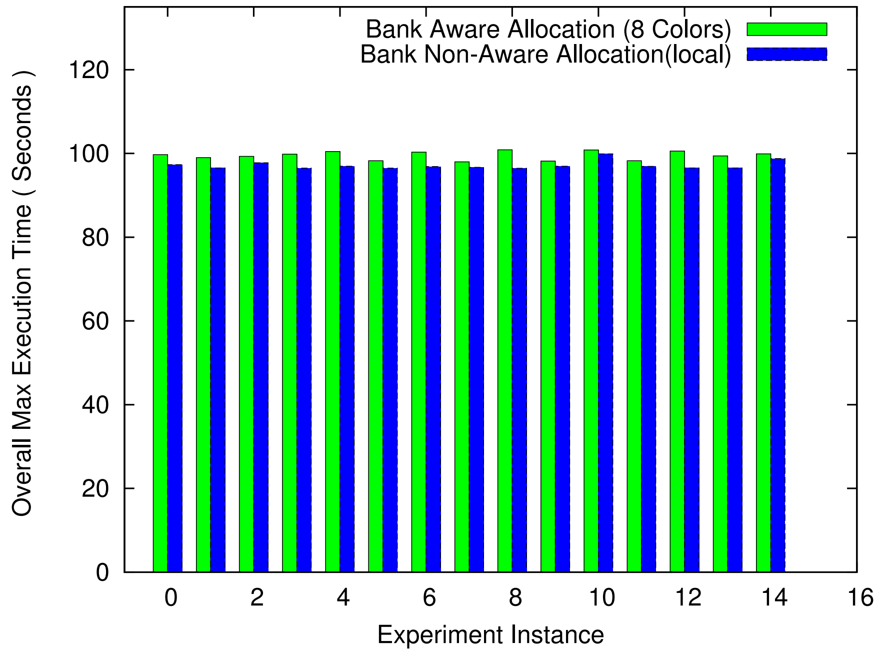


Figure 2.9 Composite Benchmark (8 Threads, 10 Pages per Thread)

Table 2.1 Standard Deviations for modified IS and STREAM

Benchmark	**4 Banks**	**8 Banks**	**Controller-local**
Modified NAS IS OMP	0.06824	0.06272	0.866052
Modified STREAM	2.950072	3.919421	12.63999

Table 2.2 Standard Deviation for Composite Benchmark Execution

Configuration	**2 Banks**	**4 Banks**	**8 Banks**	**Controller-local**	**Opposite-controller**	**1 Node**	**2 Nodes**	**Circular-allocation**
32 Threads,2 pages	0.39536	0.57046	0.02552	0.07569	0.22504	0.12855	0.04264	0.03664
8 Threads,10 Pages	–	–	1.13342	0.67696	–	–	–	–
32 Threads,10 pages	–	–	0.38644	0.19679	–	–	–	–
48 Threads,10 pages	–	–	2.23573	0.38127	–	–	–	–

One is bank address hashing. In case of non-bank aware controller-local allocation, we perform 8 kB allocations using Tilera's API that allocates at 64 kB (page size) granularity. Effectively, we only access the first 8 kB of each page, for which the bank is randomized due to bank address hashing. Our colored allocation restricts thread 0 in each quadrant to bank 0 of the closest controller and thread 1 to bank 1 of the closest controller, effectively restricting accesses to 2 banks. But in case of non-colored allocation, due to bank address hashing, the first 8 kB of the page (which we access) can be any bank (from 0-4), so there is a high probability of more bank level parallelism.

The second, less probable reason is that the number of threads per controller is less than the number of MDN ports per controller. There are two threads per quadrant and 3 MDN ports per controller. Therefore, any delay due to serialization at ports is avoided.

Table 2.1 shows the standard deviation of the execution times of IS and sustainable bandwidth reported by STREAM ADD kernel over 15 executions. We can observe that bank-aware allocation results in tighter bounds for IS and STREAM. Table 2.2 shows the standard deviation over 15 experiments corresponding to the plots of each experiment having *5 iterations* for Composite Benchmarks. We observed that for low memory footprint, the allocator yields tighter bounds. Bank-level parallelism improves performance for higher memory footprint by increasing the bandwidth (see Figures 2.6 and 2.7), but at the same time we also observe increased jitter (see Table 2.2) compared to controller-local allocation except for 8 banks. We can also observe that as we increase the number of threads from 32 to 48, the jitter increases as we change the number of threads from 8 to 32. The decrease of jitter is likely due to increased bank-level parallelism. But as we increase the number for threads to 48, the increased NoC contention and bank sharing counter the performance improvement. For controller-local allocation, a higher level of serialization takes place as we increase the memory footprint, likely causing a depreciation in performance with reduced jitter.

2.5 Related Work

The idea of DRAM and cache partitioning for maximizing memory throughput and performance isolation at the OS level is explored in several works. In Jantz et al. [Jan13], the primary objective is to optimize the power consumption and bandwidth by consolidating and relocating pages. This allows the unused parts to be put in a lower power state. The work also introduces an abstraction of application-level memory coloring, thereby introducing a degree of performance isolation for the application. This can be supported by NUMA APIs but there is no provision for bank-aware or rank-aware or interleaved allocation. A more closely related work is Jeong et al. [Jeo12]. It proposes to optimize power consumption, increase memory throughput and improve performance isolation by means of bank-level partitioning. The OS is modified to restrict accesses from a given thread/core to a given bank, thereby isolating access streams. Since bank restrictions reduce the number of banks available per thread, sub-ranking is employed to improve memory bandwidth memory. This is done by breaking a rank (64 bit) into multiple sub-ranks (8, 16 or 32 bits). Each sub-rank has an independent set of banks capable of holding different rows. This effectively increases bank level parallelism but does not consider multiple controllers. There are a few key differences. First, they operate at kernel level rather than in user space. This has the advantage of seamless integration into an application. Second, a rank is explicitly considered for increasing bandwidth while our focus is on banks and controllers. The third difference is due to the platform. Instead of actual hardware, the evaluation is based on a simulator. Fourth, the effect of the NUMA allocation policy is not considered. Park et al. [Par13], another software-based approach, propose to increase memory bandwidth and reduce cross interference in a multicore system by dedicating multiple dedicated banks to different cores. Page allocation within the dedicated banks is randomized to reduce row-buffer conflicts and to further minimize unwarranted cross-interference. Due to the implicit assumption of more banks than cores, this approach excels when the number of banks is much larger than the number of cores. It can also be observed that none of the approaches focus specifically on improving WCET.

Since multicore systems have multiple memory channels, memory channel partitioning is one potential solution to improve performance isolation. Muralidhara et al. [Mur11] propose an application-aware memory channel partitioning. They consider partitioning memory channels and prioritized request scheduling to minimize the interference between memory-bound tasks from cache bound tasks and CPU bound tasks. The drawback of this approach is that it does not consider the priority of tasks. Hence, a hard deadline cannot be guaranteed by this approach.

Apart from software based approaches, there are multiple works on improving predictability of memory controllers. PRET [Rei11] employs a private banking scheme, which eliminates bank sharing interference. Wu et al. [Wu13] take a similar approach but both approaches differ in scheduling policy and page policy. There are several other works closely related to our work [Ake07],[Goo13],[Pao09]. AMC [Pao09] focuses on improving the tightness of WCET estimation by reducing interference via bank interleaving and reducing inferences using a close-page policy. The drawback of this proposal is that instead of treating banks as resources, it treats memory as a resource. Akesson et al. [Ake07] present a similar approach to guarantee a net bandwidth and provide an upper bound on latency. They use bank interleaving to increase the available bandwidth. For bounding latency, they use a Credit-Controlled Static-Priority arbiter [Åke07].

The drawback of this approach is also is same as that of AMC.

Goossens et al. [Goo13] presents a proposal to improve the average performance of hard and soft real time system(s) on a FRT (firm realtime) controller without sacrificing hard/firm real time guarantees. It utilizes bank interleaving and proposes a new conservative closed page policy to maintain locality within a certain window. The drawback of this approach is that it does not eliminate bank conflicts completely.

Caches also impact performance isolation and there are several studies regarding the same at both hardware and software levels[Kim04],[Nes07],[Lie97],[Lin08],[Zha09],[Soa08],[Din11], [War13],[Man13]. The basic idea behind software-based approaches is cache coloring. Cache sets are considered as resources and are managed either statically or dynamically. Mancuso et al. [Man13] suggest a combination of profiling, coloring and lock-down to maximize deterministic allocation of hot (frequently accessed) pages, thereby reducing interference due to inter-core cache sharing. There are a few short-comings to this approach. First, this solution is built on the assumption that an identifiable memory access pattern exists. Second, it limits the cache availability due to lock-down. Third, it does not consider variable memory workloads. Fourth, it does not reduce inter-task interference if multiple tasks share the same core.

Ward et al. [War13] present a solution to reduce WCET and improve predictability for highly critical tasks using cache coloring, cache locking and cache scheduling. Heuristics are used for coloring, making this approach more flexible than cache partitioning. Ding et al. [Din11] present a way for performance improvement by reducing last level cache thrashing. They limit the cache pollution by mapping OS buffers to certain cache regions. CAMA [Her11] is a constant time dynamic memory allocator, which eliminates unpredictable cache pollution to improve predictability and to yield better WCET analysis. It achieves this by performing cache-aware splitting and by using coalescing techniques. The shortcoming of this approach is that it does not handle multilevel caches. Liedtke et al. [Lie97] present an application-transparent cache partitioning approach, thereby improving predictability and performance.

Our work primarily focuses on the impact of bank-aware and controller-aware allocation in a NoC architecture, which makes it significantly different from prior studies to the best of our knowledge.

Chapter 3

Programming Model Assessment

3.1 Background

3.1.1 NoC Architectures

NoC-based manycore architectures like Tilera [Tili] or Intel Xeon Phi [Mic] provide computing power via multiple cores. All the data transfers typically occur over the NoC in form of the messages. These messages are packetized with the required flow control or routing information and are then routed across the network to the required component. Tilera, Xeon Phi and other manycore architectures utilize 2D on-chip interconnects (mesh- or ring-based) instead of traditional bus-based architectures.

3.1.1.1 NoC

On-chip networks, also known as network-on-chip or NoC, replace the traditional bus topology with on-chip networks. This means all communication, I/O transactions and memory transactions occur via the NoC in the form of messages. Depending upon the architecture in question, we may have a single or multiple NoC interconnects available primarily to reduce contention. For example, Adapteva features three networks [Ada], Tilera TilePro64 features 6 networks and Intel SCC [Sccb] features a single network. Therefore, in such architectures efficiency of parallelization is constrained by network contention and memory contention. Unlike a traditional system bus, the NoC is generally a switching network where data is split into packets routed across the network. For example, the Tilera architecture supports a 2D mesh with wormhole (X/Y dimension ordered) routing.

3.1.1.2 Cores

Cores generally interact with the network via switches using input and output queues. For example, in the Tilera architecture, each core or tile is composed of a processor engine comprised of the CPU, cache and a switch engine, which has multiple I/O queues to route packets, i.e., connecting the core to other cores and I/O devices.

3.1.1.3 Switches

Each switch is part of the crossbar switch with multiple input and output queues that can be configured to be connected. Wormhole routing dynamically creates such connections based on a packet destination.

3.1.2 Tilera Architecture

The Tilera architecture implements a 2D mesh, also referred to as iMesh interconnect, with specialized hardware, register mapping, multiple networks that provide high bandwidth and low latency communication channels. The Tilera architecture has six mesh networks (IDN, UDN, CDN, IDN, TDN, and STN). IDN (I/O dynamic network) and UDN (User Dynamic Network) are two architecturally-defined networks for routing data among I/O controllers and tiles. MDN (Memory Dynamic Network) is a network for memory data. CDN (Coherence Dynamic network) is used for cache coherence traffic. TDN (Tile Dynamic Network) usage is similar to MDN but it serves requests for tile-to-tile transfers. The static network is used for data transfer between tiles with fixed routings.

Of these networks, only UDN is accessible to users via APIs and it does not require system intervention. The NoCMsg MPI abstraction [ZM14] is based on IPC (Interprocess communication) using UDN, where the load is distributed by appropriate domain and sub-domain decomposition and task/thread placement. The Tilera architecture has a ccNUMA architecture with per-core L2 cache and a global soft L3 cache (composed of multiple L2 caches) as the NUMA policy is used for page placement and a MESI-style protocol is used for maintaining cache coherence.

3.1.3 Architectural Implications

Consider task parallelism with fine-grained domain decomposition to maximize core utilization of nodes combined with thread-level decomposition using shared memory within a node. This requires domain decomposition and sub-domain decomposition.

Any parallel application involves computation and communication. The primary objective of this work is to reduce the communication overhead. In case of a modern HPC system composed of nodes, we mainly focus on topology-aware task placement and domain decomposition driven by interconnect bandwidth and communication pattern. Then, depending on processor architecture, I/O features, shared memory design (NUMA vs. ccNUMA, L1, L2 and L3 cache), we can decide on coarse or fine sub-domain decomposition.

Due to architectural considerations, we have following major constraints limiting parallelization:

- False Sharing;
- DSM protocol overhead (explicit data homing or hash-for-home, described in more detail below);
- ccNUMA architecture imposing latencies and bandwidth;
- NoC contention.

These factors are closely interrelated but we will analyze them one by one. First, false sharing can generate a significant amount of coherence traffic that may cause NoC contention. False sharing between cores will occur when data within a cache line is accessed in disjoint chunks by multiple threads, i.e., without actual data sharing. Furthermore, if we assume random thread placement, there is a high chance that data is scattered across tiles, and subsequent accesses lead to invalidation and cache coherence traffic across the entire NoC. On the Tilera architecture, this can be triggered by hashing a page across tiles at cache line granularity. To avoid this, proper data and work division among tasks and appropriate work sharing among threads is required. Further, proper tiling of loops considering L1, L2 and L3 caches will increase the cache hit rates.

The second factor is data homing. There are three ways to home data in the Tilera architecture: Local homing, remote homing and hash-for-home [Tild]. 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 DDR 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. In general, if the access pattern cannot be determined, hashing tends to yield a better performance due to effective load balancing across

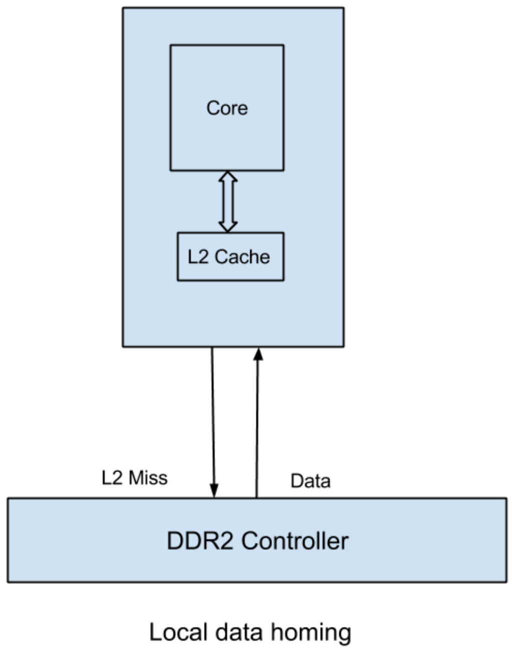


Figure 3.1 Local data homing

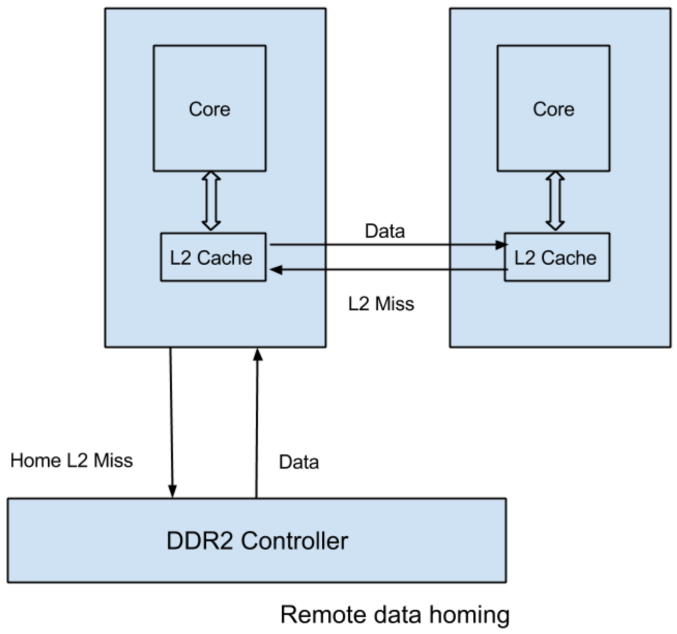


Figure 3.2 Remote data homing

the set of tiles. But in case of the hybrid model, it will lead to performance deterioration and unintended network contention. Alternatively, the Tilera architecture provides APIs for explicit control of data homing at the page level. For example, if a page is shared across all tiles, it is possible to just hash that particular page across tiles. Data can be dynamically rehomed upon task migration as well. But it is often better to explicitly control data homing or use local homing. This will help in localizing the traffic within a set of tiles associated with the process and thus reducing the NoC contention.

The third factor is a ccNUMA (cache coherent NUMA) architecture. The Tilera architecture has four memory controllers and is a cache coherent NUMA architecture. There are two ways in which page allocation is performed. If memory striping is enabled, only one controller is visible to Linux and the page (64 kB size) is striped across the memory controller at 8 kB granularity. If memory striping is disabled, all four controllers are visible and the page is allocated at the closest memory controller. Since it is a cache coherent NUMA architecture, coordinated homing of data along with task/thread placement can lead to a reduction in network traffic and better load distribution over the NoC.

An even load distribution among networks is key to maximizing performance. On one extreme, the OpenMP model utilizes shared memory parallelism, where all the networks (except UDN) are utilized. On the other extreme, we have process-based parallelism, which relies on MPI messages sent over UDN for data exchanges. The level of utilization of NoC meshes varies. But in case of the hybrid model, the impact of domain decomposition and sub-domain decomposition among threads within a process has an impact on performance.

Task/thread locality also plays a major role in reducing NoC contention. Appropriate placement of task(s) or thread(s) can reduce the latency in communication and NoC contention.

Another factor to maximize performance in any model is the pinning of processes and threads. Under NUMA policy, it is critical that threads or processes do not migrate. By architectural requirements, if we use UDN, we need to set the affinity of a process to a particular tile as it is used for the routing of packets. Processes cannot migrate, but for threads there is no such constraint, so we need to define affinity of threads appropriately and explicitly.

Considering all of the above factors, domain and sub-domain decomposition is non-trivial.

3.2 Design

The key driver behind utilizing UDN for providing an MPI-like abstraction is that it is extremely low latency and can be realized at the user level, i.e., it can be used to stream data and messages without any operating system involvement.

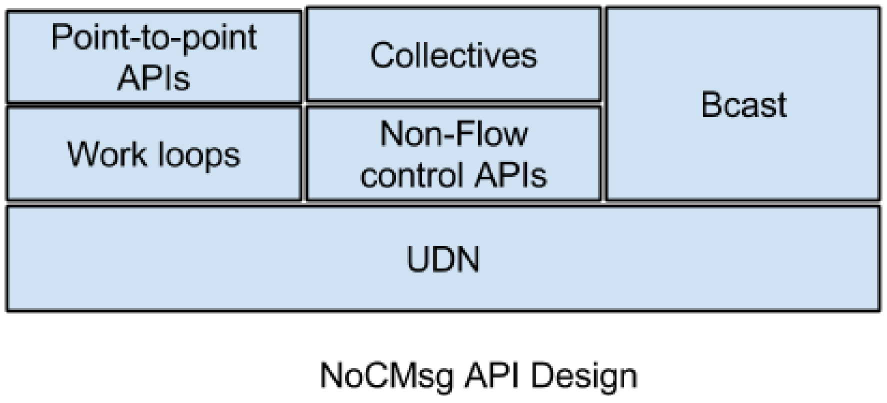


Figure 3.3 NoCMsg API design

There are two ways to design an MPI-like abstraction using UDN, interrupt based and polling based. iLib [Wen07a] 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 [Kan09] is built around iLib using primitives to support more complicated collectives like all-to-all, scatter-gather etc.

The existing NoCMsg library [ZM14], 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:

- trysend: keeps polling the send queues to check for any pending send;
- 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. There are separate queues for handling known and unknown receives.

Fig. 3.3 and Fig. 3.4 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 [ZM14].

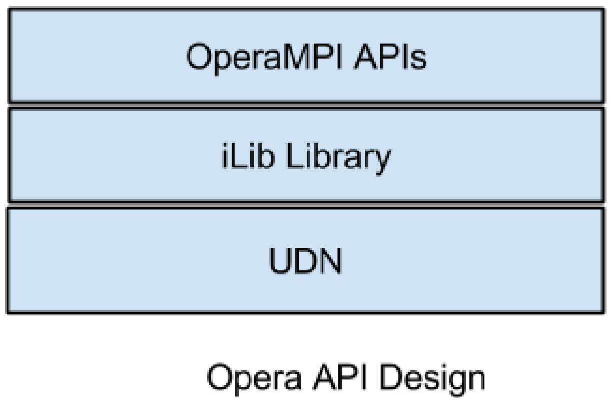


Figure 3.4 OperaMPI API design

3.3 Implementation

The implementation effort consisted of extensions to the NoCMsg library.

3.3.1 NoCMsg

The current implementation of NoCMsg supports the original set of NAS benchmarks but since they are not hybrid, they cannot be utilized for our assessment. We identified several C and FORTRAN benchmarks supporting the hybrid model of execution. Our initial set consisted of CoMD [Com], the NAS Multi-Zone Suite [WJ03], CoSP2 [Cos], HPGMG-FV [Ada14], and LULESH2.0 [Kar13]. We investigated the level of modifications required in the current NoCMsg library to support these codes, which resulted in adding the following features:

- Support for DOUBLE_INT;
- support for MINLOC and MAXLOC reductions;
- support for 64 byte unsigned int (uint64);
- support for byte datatype;
- sendreceive functionality;
- MPI_Get_count; and
- MPI_Test.

UDN APIs only support data transfer in terms of 4 byte unsigned integers, therefore, a forward and reverse mapping of data types needs to be maintained. This mapping was updated to support the needed data types. While updating the mappings, we also found that mappings for byte and char need to differ or they may lead to buffer overflow problems. To avoid this buffer overflow, forward and reverse conversion functions were added and other functions were modified to use these functions instead of directly accessing the mappings. This also increases the robustness of the APIs.

The introduction of the new data types was made fairly easy via abstractions over data type mappings. Support for DOUBLE_INT and uint64_t type was required by the benchmarks, so the reduction implementation was extended to cover the new data types. MINLOC and MAXLOC reduction support was also added as part of the extension.

A few benchmarks required Sendreceive functionality, which was layered over Irecv, Isend and Wait of the NoCMsg API.

MPI_Get_count, used by CoMD, was designed and implemented using the conversion functions added to support new data types and via extensions of the internal data structures. MPI_Test was also implemented along similar lines with MPI_Get_count functionality.

The NAS Multi-zone and HPGMG-FV required communicator split functionality. All the APIs and internal functions were modified to handle global and local ranks within a communicator appropriately. Sanity checks in the Isend, Irecv and Wait functions were added to handle inappropriate request handles. We also found that additional flow control, where a blocking send waits for a receiver to acknowledge the data reception before send unblocks, was violating MPI semantics, so this mechanism was removed completely. Since UDN ensures a lossless packet transfer, it is safe to assume once the packets are queued for sending they will reach the destination, so any additional software-based control is not required. Send receive matching logic was updated to only consider the source and tag.

3.3.2 Benchmarks

Two major criteria were considered for selecting a benchmark:

1. Flexibility in terms of process and thread configuration; and

2. The per-thread input should fit within the LLC (Last Level Cache) of each tile.

Flexibility in terms of process and thread configuration allows us to make a more accurate assessment of the communication overhead involved. If we can fit the per-thread input within the LLC, cold misses will bring data into LLC, but no other DRAM accesses will occur from there on. This ensures that we will measure the effect of the NoC on scalability instead of bandwidth limitations on DRAM. Keeping these two objectives in mind, benchmarks were chosen (discussed in detail in the Section 3.4). The modifications to benchmarks selected for assessment were minimal. We replaced the MPI constructs with NoCMsg constructs and modified the tag values of sends and receives to be within the supported range.

We also needed to distinguish between thread and process affinity. Due to architectural constraints, the usage of UDN mandates that the affinity of a process, once set to a particular tile, has to be maintained. When this process spawns threads, they inherit the affinity mask of the parent. Thus, all the threads contend for the same CPU. Using the GOMP_CPU_AFFINITY environment variable, a partial solution is possible, but since it is a system-wide setting, it still led to CPU contention when multiple processes were spawning threads. After investigating libgomp (the Linux OpenMP library), we found that using a Pthread affinity call along with the Tilera affinity call, we were able to utilize all CPUs and alleviate the contention. But this required modifying benchmarks to set the affinity of OpenMP threads as they are launched. Thus, explicit calls to set the affinity were added to every OpenMP section as this affinity mask was not persistent.

3.4 Evaluation

3.4.1 Hardware Platform

We utilized the Tilera TILEPro-64 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.

3.4.2 Key Ideas

As mentioned previously, the aim of this work is to analyze the three programming models with respect to the communication overhead under various configurations. This means we are indirectly trying to analyze the latencies due to NoC(s). To do this, we need to minimize the LLC cache misses so as to minimize the references to the memory.

Since we need to analyze 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. 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 in increments of 8. 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 from 8 to 56 in increments of 8. For MPI (NoCMsg) only, we vary the number of MPI tasks from 8 to 56 in increments of 8 and fix the number to threads per MPI task to 1. For the hybrid case (NoCMsg + OpenMP), two configurations were used. In one hybrid configuration, we fix the number for MPI tasks to 8 and vary the number of threads per task from 1 to 7 in increments of 1. In the second hybrid configuration, we fix the number of MPI tasks to 16 and vary the number of threads from 1 to 3 in increments of 1. As with weak scaling, we modify the input sizes to fit within the L2 cache(s).

This approach minimizes all other latencies, e.g., latencies due to memory, which would otherwise dominate and skew the assessment. This exposes the overheads due to the NoC. By fitting data within LLC, we can study the impact of scaling on a many-core with good locality and minimize other latencies.

To assess the impact of the DSM (Distributed Shared Memory) protocol at L3 level, experiments are conducted with and without hashing. As discussed in Section 3.1.3, 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 [ZM14], we can often get better performance with UDN (NoCMsg only) or with UDN+MDN (hybrid model).

The MPI-only and OpenMP-only configurations use the default sequential task/thread mapping, but for the hybrid model, a different mapping is used to group threads within a process in close proximity. Fig. 3.5 shows the hybrid mapping for 8 MPI tasks. We place the communicating threads per MPI task in close proximity (in this case in the same row) to localize contention. For 16 MPI tasks, the mapping was derived from the 8 MPI task mapping but the CPU set per

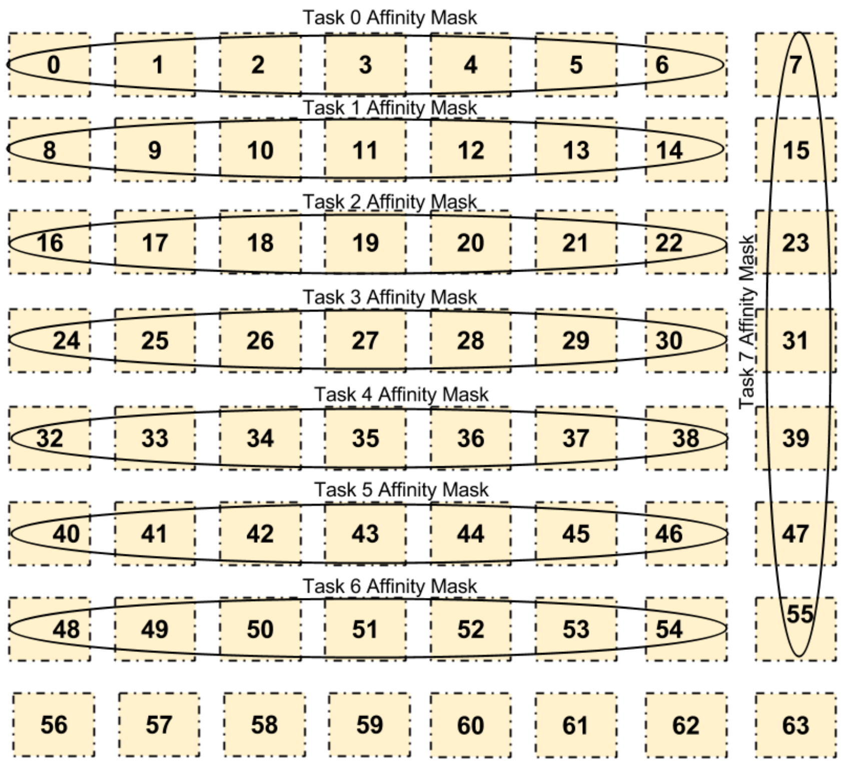


Figure 3.5 CPU Affinity used for Hybrid execution (8 MPI tasks)

task with 16 MPI tasks has half the number of threads (cores) as that for of 8 MPI tasks.

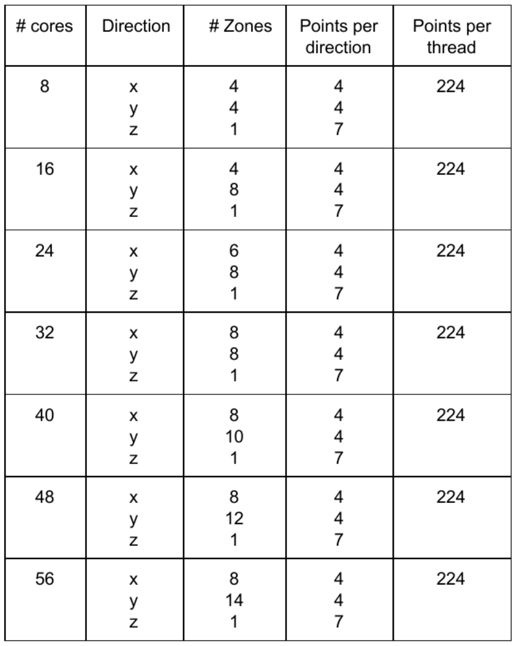
3.4.3 Benchmarks

LULESH2.0, CoSP2, HPGMG-FV, CoMD, LU-MZ, SP-MZ and BT-MZ comprise the initial set of benchmarks as they support the hybrid programming model. By varying the number of MPI tasks and threads per MPI task, we can experiment with all three programming models using the same code base. After analyzing data decomposition in the benchmarks and the resulting memory footprint, HPGMG-FV and CoSP2 were ruled out due to their high memory footprint. LULESH was ruled out due to constraints on the number of MPI tasks. It only supports numbers of MPI tasks in a perfect cube, and since we only utilize up to 56 cores, experiments for the MPI only model would be too restricted.

CoMD 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 number of local boxes per thread comes down to 8, and with a maximum size of 64 atoms per box, the memory footprint for the data structure containing information on atoms amounts to about 44 kB.

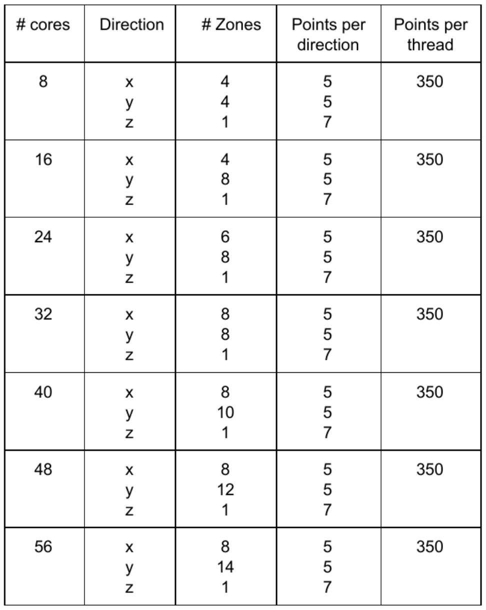
NAS Multi-Zone benchmarks, derived from the original NPB Suite, divide their 3D mesh input into multiple meshes or zones. Each zone is solved independently. After each iteration/time step, boundary values are exchanged between zones. In case of the MPI or OpenMP model, each MPI task/thread is assigned 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. This suite has three benchmarks: LU-MZ (Lower-Upper Symmetric Gauss-Seidel), SP-MZ (Scalar Penta-diagonal) and BT-MZ (Block Tri-diagonal) derived from NPB. In case of SP-MZ and LU-MZ, partitioned zones are identical in size but in case of BT-MZ, the size of each successive zone increases. Assuming 24 double precision variables per mesh, the memory footprint per thread is calculated to be approximately 42 kB for SP-MZ and approximately 65 kB for LU-MZ [WJ03].

These benchmarks do not restrict the number of tasks given an input size, which allows us to analyze performance variation and scalability over varying number of cores under different programming models. For NAS Multi-zone benchmarks, the default input classes do not support weak scaling. Instead, we derived new problem sizes for weak scaling of SP-MZ and LU-MZ based on CLASS S by varying the number of zones and keeping the points per zone constant.



# 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

Figure 3.6 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

Figure 3.7 LU-MZ Input Specifications

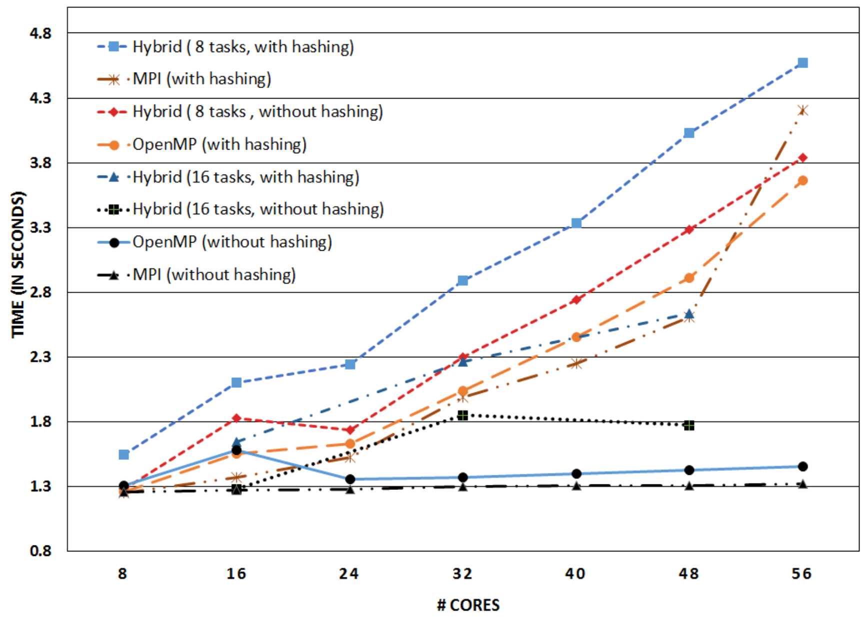


Figure 3.8 Average runtimes of LU-MZ Weak Scaling

3.5 Results

The evaluation assesses the performance of benchmarks using communication via UDN versus 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. All the experiments are repeated 15 times to get statistically viable data. Apart from the data generated by our experiments, we also used data presented by Zimmer et al. [ZM14] and Serres et al. [Ser11] in our analysis.

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. 3.8, Fig. 3.9 and Fig. 3.10 show the average execution times of the three benchmark(s) in three different programming models, each with hashing turned on and off.

The legends of all the figures are top-down in order of the y-axis metric for the largest core configuration, e.g., in Fig. 3.8, hybrid (8 MPI tasks) has the worst performance for 56 cores.

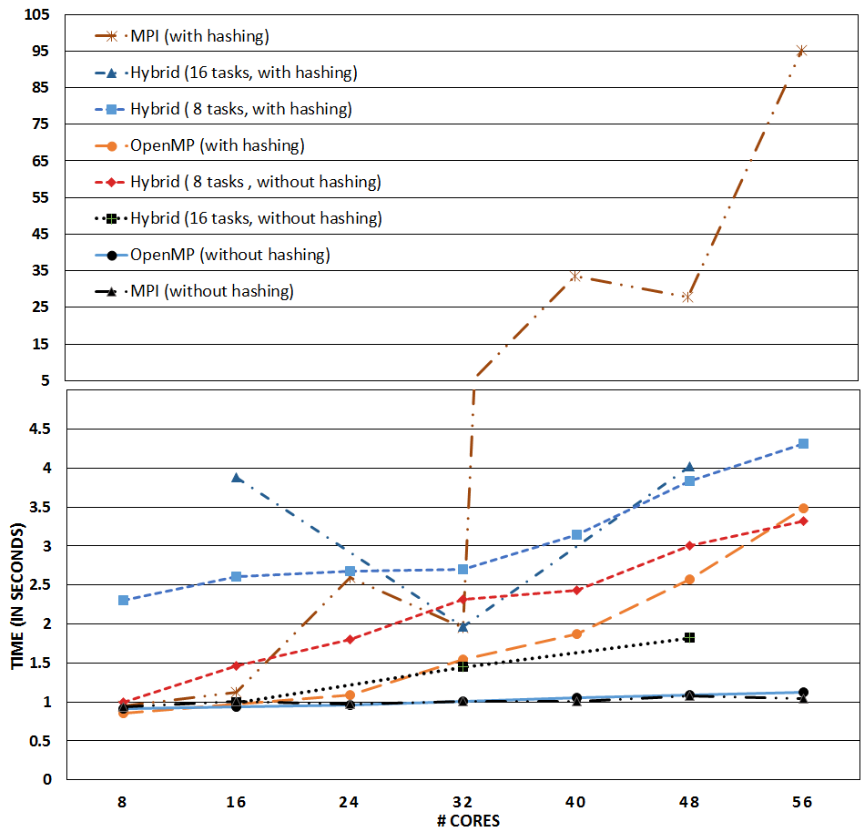


Figure 3.9 Average runtimes of SP-MZ Weak Scaling

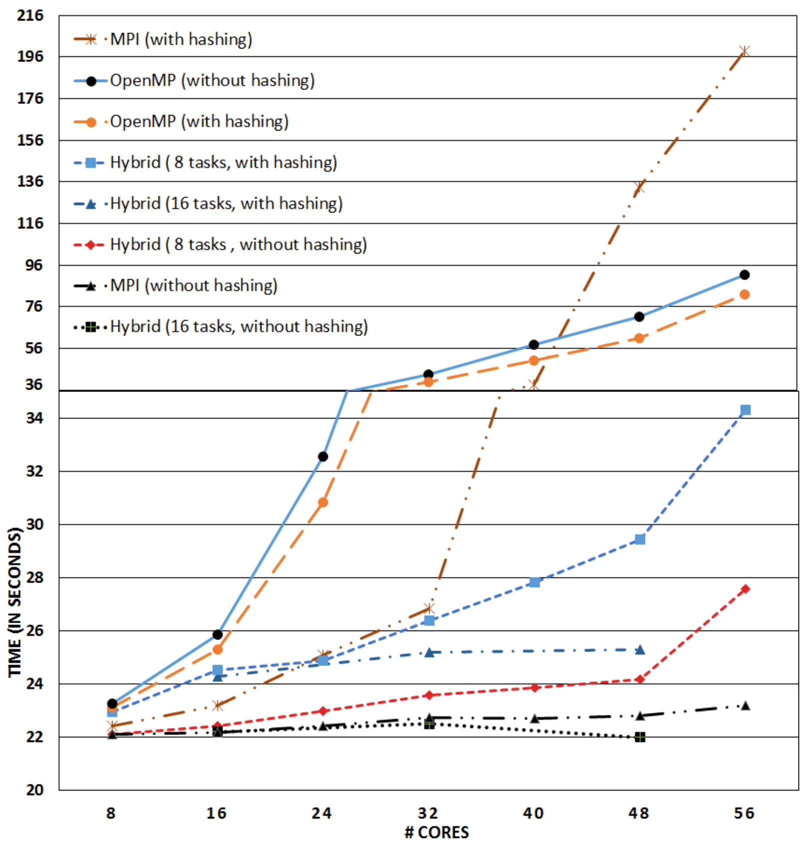


Figure 3.10 Average runtimes of CoMD Weak Scaling

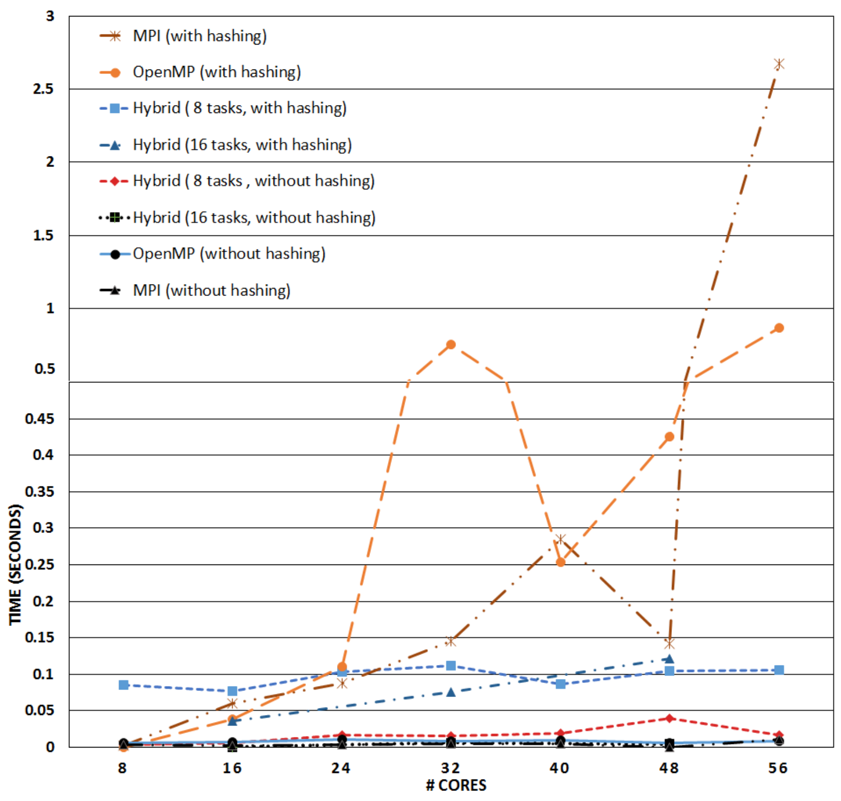


Figure 3.11 LU-MZ Standard Deviation corresponding to weak scaling

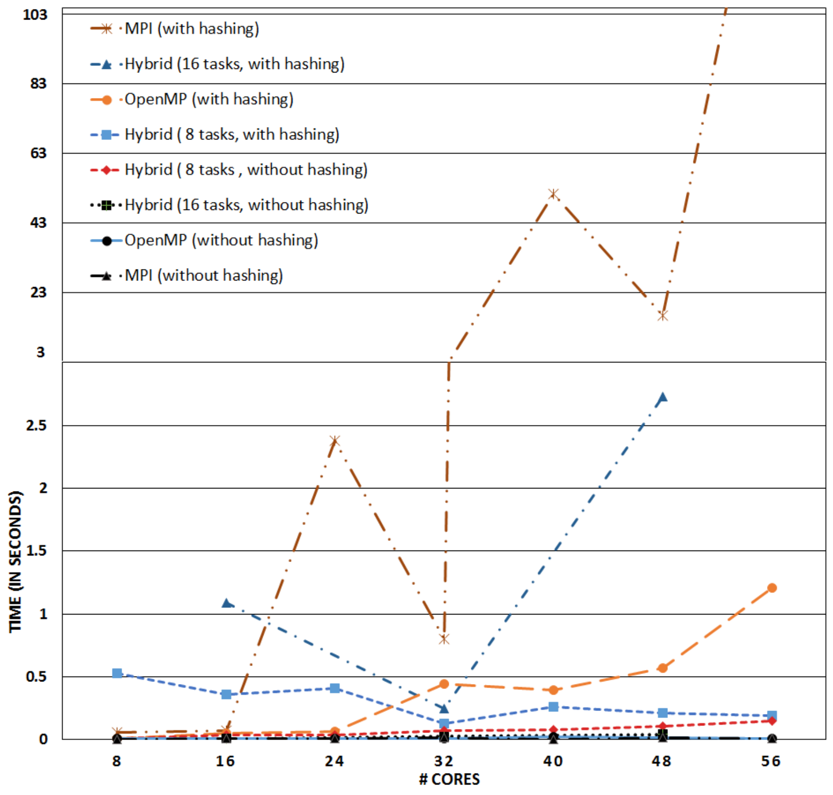


Figure 3.12 SP-MZ Standard Deviation corresponding to weak scaling

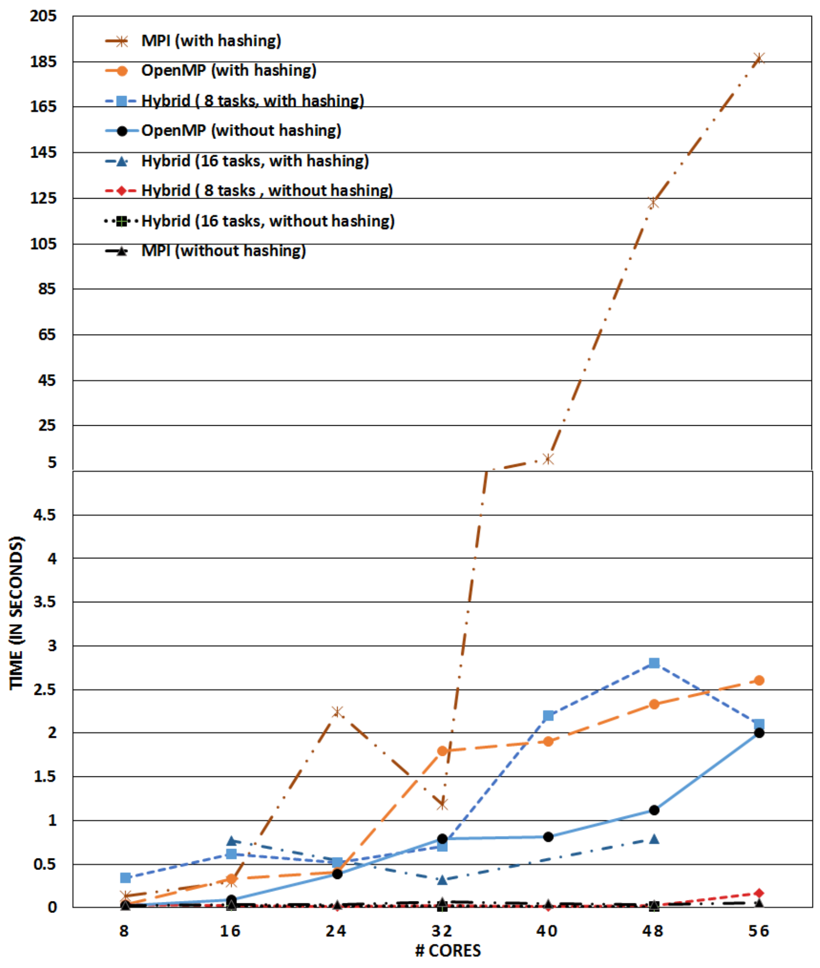


Figure 3.13 CoMD Standard Deviation corresponding to weak scaling

We observe from the Fig. 3.10 that the execution time is generally high compared to SP-MZ and LU-MZ. Even though we have reduced the local process/thread data, the halo data still exceeds the cache, which results in increased off-chip accesses at certain stages of processing.

We can see from the plots that the MPI-only and OpenMP-only programming models scale well with hashing turned off. Even though MDN has twice the bandwidth of UDN, MPI manages to outperform OpenMP most of the time when hashing is turned off. Consider Fig. 3.9: Up to 32 cores, OpenMP either performs better than MPI or matches the MPI performance. But after 32 cores, we observe a slight deviation of OpenMP from MPI. Memory networks are pull-based and require additional messages per transfer but UDN messaging is push-based. Due to this, for smaller transfers, we observe low latency but for larger transfers, UDN performs better in-spite of its lower bandwidth (consistent with Zimmer et al. [ZM14]). In the case of SP-MZ, MPI-only surpasses OpenMP-only at 32 cores, when contention due to MDN overheads results in better performance of MPI-only. In case of LU-MZ, this point occurs at 8 cores, after which MPI performance is better than OpenMP. CoMD OpenMP-only always shows inferior performance compared to MPI and Hybrid when hashing is off. This is due the fact that CoMD does not take advantage of the ccNUMA architecture in OpenMP-only execution. The default setting is to home the dynamically allocated process private pages at the tile/core of the process that allocated the data, instead of the core which is later accessing the page. As a result, all the data is homed at core 0 (to which the process is bound to). This leads to high latency and contention due to multiple tiles requesting data from the same home tile. For MPI-only, this default policy allows us to place the data at the local core, thus improving performance. We do not see this behavior for the NAS benchmarks as they do not perform any dynamic allocation like CoMD. The default policy of homing the stack data to the local tile and good data decomposition results in performance equivalent to MPI-only. The increased halo data for larger inputs under the same number of processes results into an inferior performance due to more computation and more off-chip accesses. The halo data also increases the computation time in certain stages of computation, as OpenMP parallelization also involves halo data in those stages.

We can observe from the plots that hybrid programming (with 8 tasks) generally performs inferior to OpenMP-only and MPI-only (except for CoMD) with hashing off. There are multiple reasons for this. The first reason is the synchronization overhead of the threads. The OpenMP sections are scattered across the program for the Hybrid/MPI code base. So every time a parallel section is encountered, threads in the pool need to be activated and deactivated. The higher the number of threads, the higher the overhead. In case of SP-MZ and LU-MZ, OpenMP parallelization is done per zone. So if we have 4 zones per process and 2 threads per process, 2 threads are activated every time a zone needs to be updated. This is contrary to the pure OpenMP implementation for SP-MZ and LU-MZ, where each thread is assigned zones and we can use temporal and spatial locality. So the potential loss of locality (which causes increased contention) and thread synchronization overhead lead to performance deterioration compared to pure OpenMP and pure MPI. In case of CoMD, we see a slight performance deterioration compared to MPI-only, as almost all the data is remotely homed. Contention and latencies get worse as we increase the number of threads. The hybrid model still performs better than OpenMP-only due to an increased number of home caches: One for all threads in case of OpenMP-only compared to one per MPI task in case of 8 MPI tasks in hybrid configuration. Hybrid results in better load distribution and localized traffic, leading to lower network contention and shorter latencies. It is still inferior to MPI-only as halo data is still high per process compared to MPI-only processes. This combined with reduced number of home caches, results in an inferior performance.

For 16 MPI tasks in the hybrid configuration, the trend observed for 8 MPI tasks in hybrid configuration remains similar with a few changes. 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, lower NoC contention and lower number of halo boxes, which lowers the off-chip memory accesses and computation cost. 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 [Tild]:

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 *all* for our initial analysis. Notice that this tends to maximize NoC contention. Since hashing distributes the data across all caches, the effective cache capacity becomes larger for small numbers of threads (compared to their aggregate L2 capacity). As shown by Zimmer et al. [ZM14], the hop count for each access varies. This increases the jitter, which is not very high if viewed per core, but as we increase the number of cores used, it accumulates potentially leading to performance deterioration. For example, if we look at Fig. 3.9 and compare the trends for OpenMP with hashing with OpenMP and MPI without hashing, we can observe that up to 16 cores, we get better or similar performance but after 16 cores, performance starts to deteriorate for hashing. For CoMD, hashing improves the performance of OpenMP-only: Instead of a single home cache, data is distributed across all tiles yielding a slight performance improvement. As observed from the Figure 3.10, the difference either increases or remains constant up to 48 cores but at 56 cores, it decreases. This is likely due to NoC contention countering the performance improvement due to hashing.

Due to NoC contention, the variance in execution time is high compared to hashing off (see Figures 3.11, 3.12 and 3.13) and increases as we increase the number of cores. Almost all configurations show an increase in the jitter/standard deviation (SD). MPI-only is the most affected and the hybrid configurations are the least affected. SP-MZ and CoMD are more affected by hashing than LU-MZ for MPI-only. To understand the reason behind resilience of hybrid execution to variance compared to MPI-only, we set up an experiment to understand the impact of different types of hashing. We ran on 56 cores with MPI-only and Hybrid varying the types of hashing. As we can observe from Figures 3.14, 3.15 and 3.16, the point of interest is the transition from *allbutstack* to *all*. For all three benchmarks, we observe a significant decline in performance for MPI-only 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 MPI-only compared to hybrid for hashing.

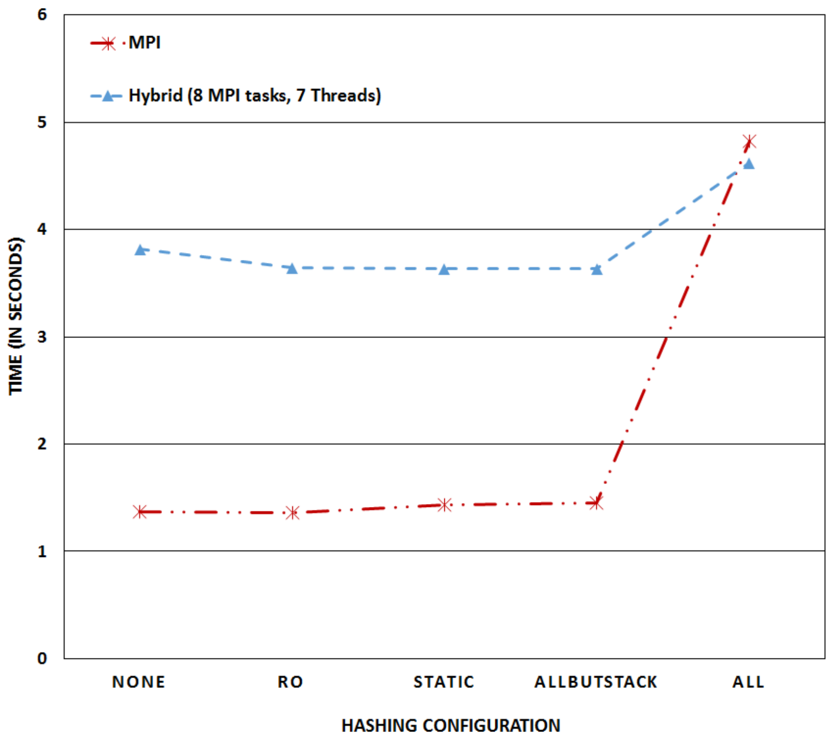


Figure 3.14 Average runtimes of LU-MZ Hashing for 56 cores

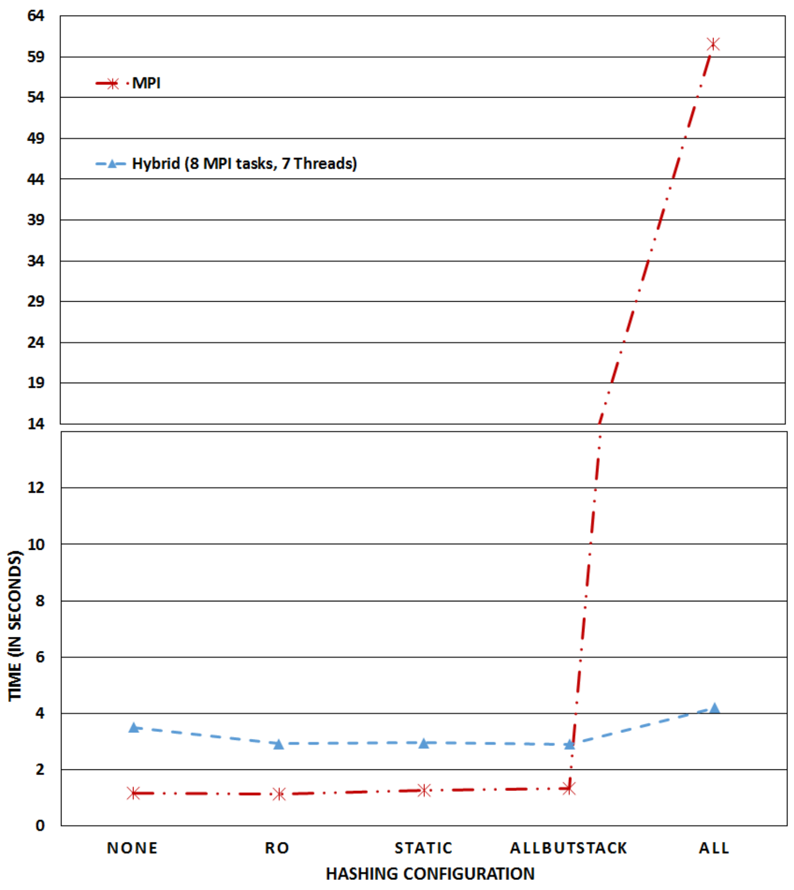


Figure 3.15 Average runtimes of SP-MZ Hashing for 56 cores

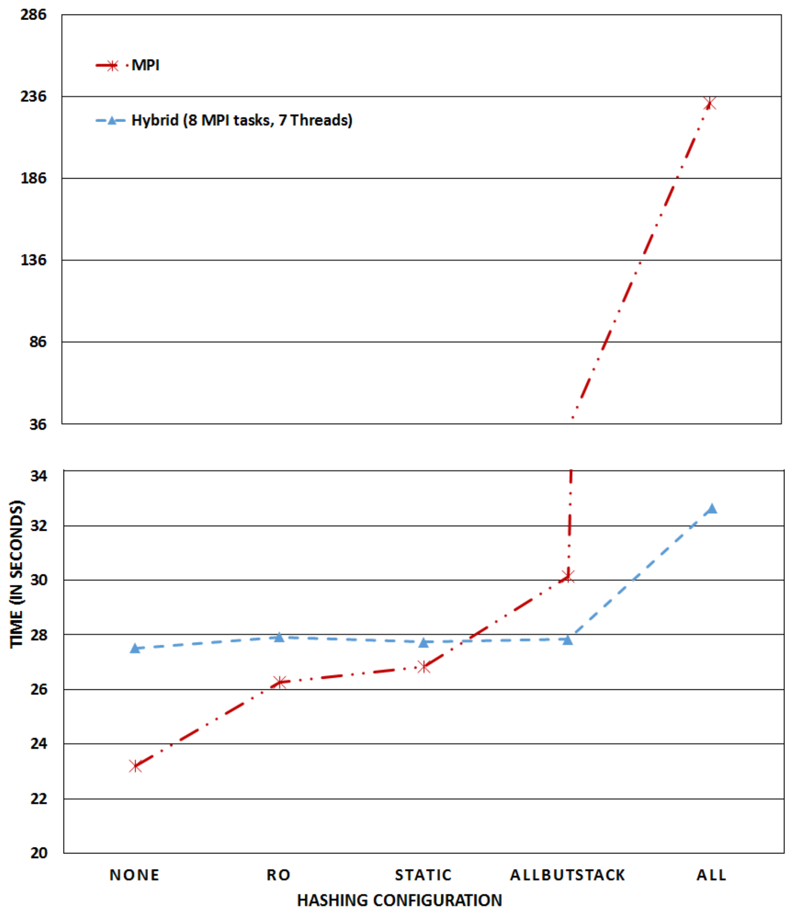


Figure 3.16 Average runtimes of CoMD Hashing for 56 cores

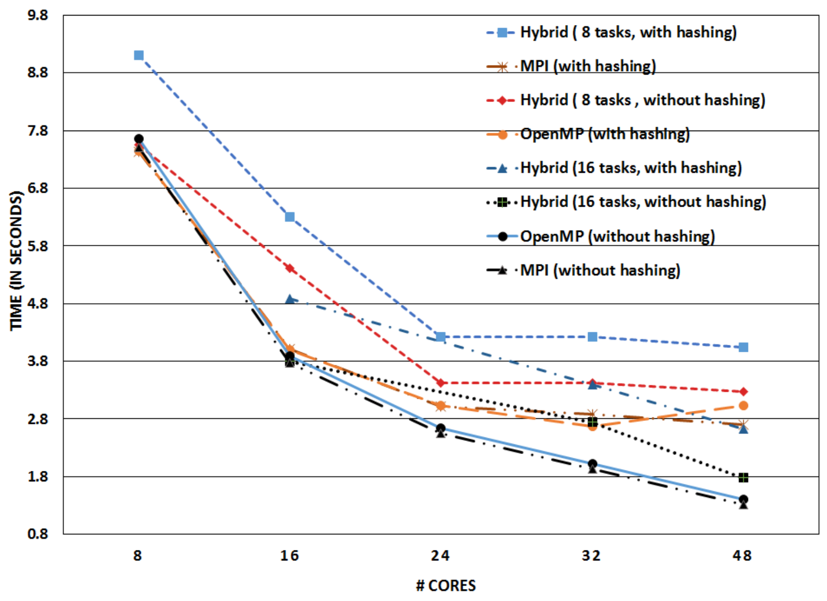


Figure 3.17 Average runtimes of LU-MZ Strong Scaling

We also conducted strong scaling experiments. For SP-MZ and LU-MZ, we use the 48 core input for strong scaling (i.e., 96 zones and number of points per zone as per Figures 3.6 and 3.7, 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). Figures 3.17, 3.18, and 3.19 show the average runtimes of LU-MZ, SP-MZ and CoMD over various processor counts, respectively. The executions for SP-MZ and LU-MZ corresponding to 40 and 56 cores are omitted for strong scaling as they impose imbalance among cores. For CoMD, the execution corresponding to 24 cores is omitted as the given input was not compatible with a 24-core process configuration.

The average execution time is on par with the expected decrease in runtime as we increase the parallelism (OpenMP, MPI or Hybrid). But any performance improvement diminishes as we increase the core count, and in some cases the performance starts to deteriorate. This is due to increased NoC contention, which counters the potential improvement. Consider Figure 3.19. For up to 32 cores, performance improves for all configurations, but after that performance deteriorates for OpenMP but even more so for MPI-only (which is in line with observations for weak scaling). We can also observe that the runtime of CoMD is generally higher compared to

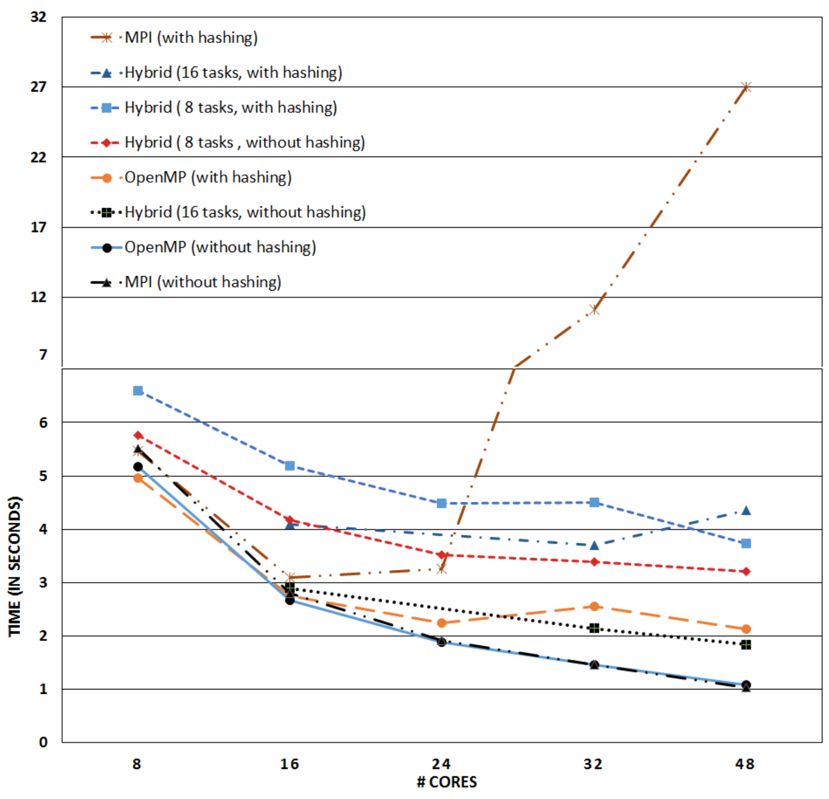


Figure 3.18 Average runtimes of SP-MZ Strong Scaling

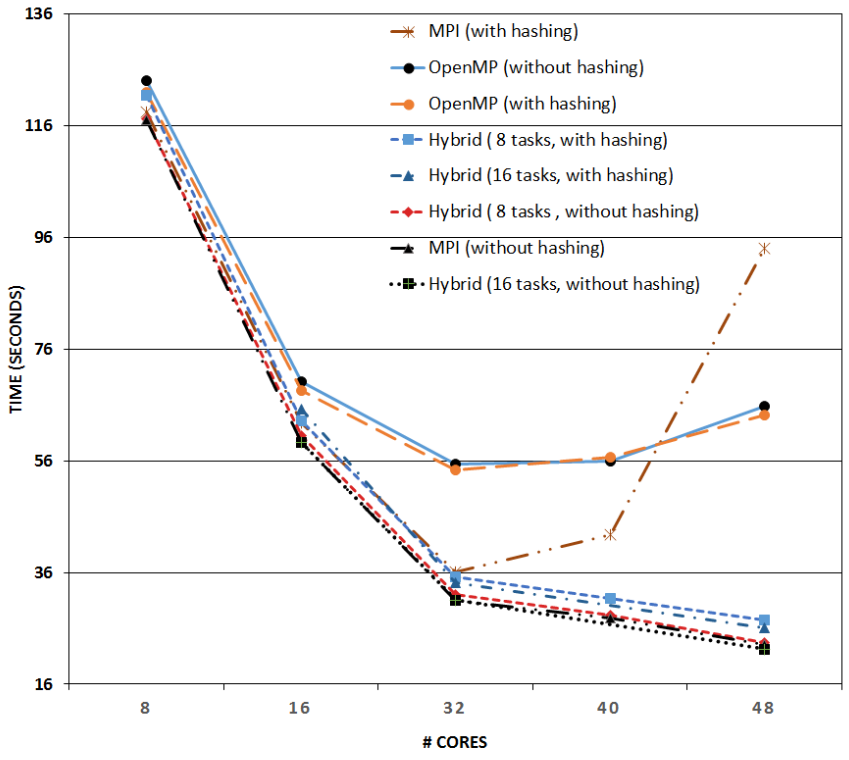


Figure 3.19 Average runtimes of CoMD Strong Scaling

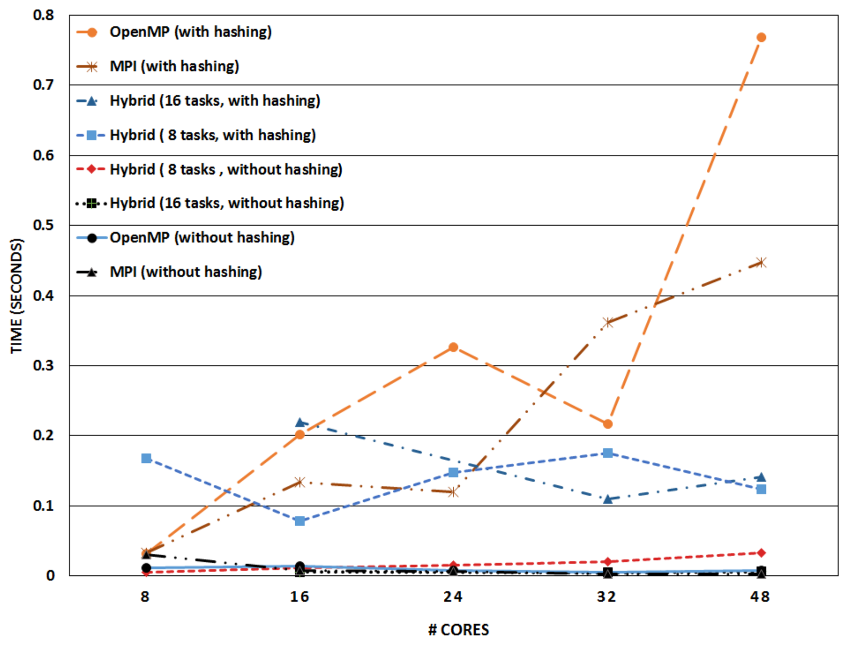


Figure 3.20 LU-MZ Standard Deviation corresponding to strong scaling

SP-MZ and LU-MZ which is in line with the observations for weak scaling due to the larger number of off-chip accesses. In case of SP-MZ (Figure 3.18), MPI-only is worst affected due to stack sharing, as was the case for weak scaling. In case of LU-MZ (Figure 3.17), the performance tends to improve (although the improvement diminishes with increasing core count), except for OpenMP-only with hashing, where we have a slight performance deterioration. This is likely due to the NoC contention countering the advantage due to reduced data processed per core.

Figures 3.20, 3.21, and 3.22 show the SD of the runtime plots corresponding to Figures 3.17, 3.18, and 3.19. We observed that jitter (SD) is high for hashed executions compared to non-hashed execution (similar to weak scaling executions) due to NoC contention. The jitter also increases as we increase the number of cores used for execution. This can also be attributed to the NoC contention but hashing amplifies this effect, resulting in higher jitter when the core count is increased for hashed executions. MPI-only is the worst affected due to stack sharing in the hashed execution. For LU-MZ (Figure 3.17), we can observe that OpenMP-only has the worst SD unlike CoMD and SP-MZ.

Contention for strong and weak scaling in SP-MZ and LU-MZ arise due to different reasons. In case of weak scaling, the input specifications are designed such that the data fits in the LLC.

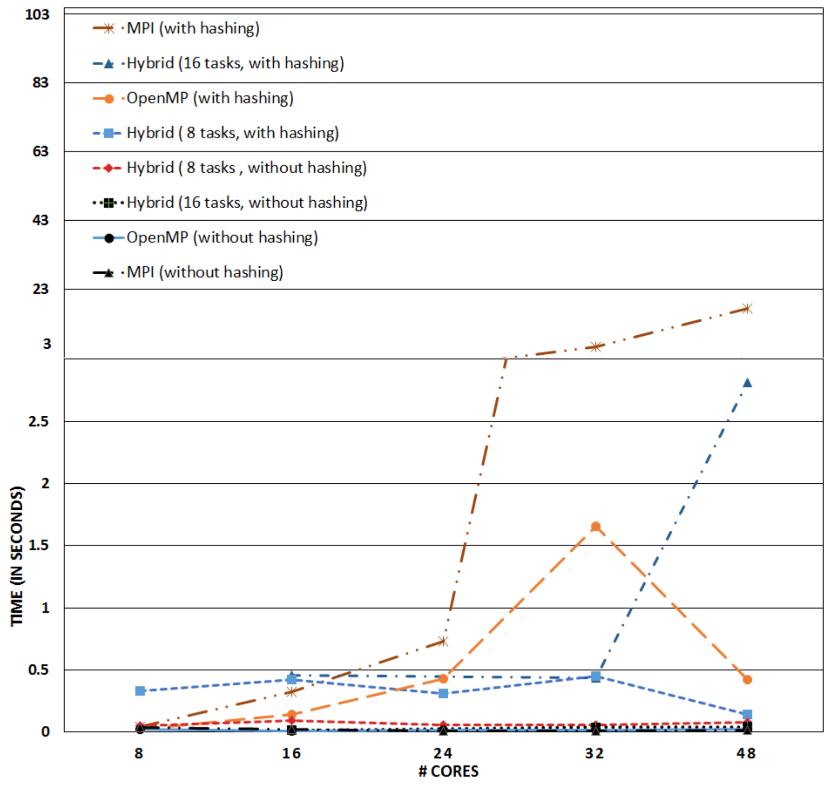


Figure 3.21 SP-MZ Standard Deviation corresponding to strong scaling

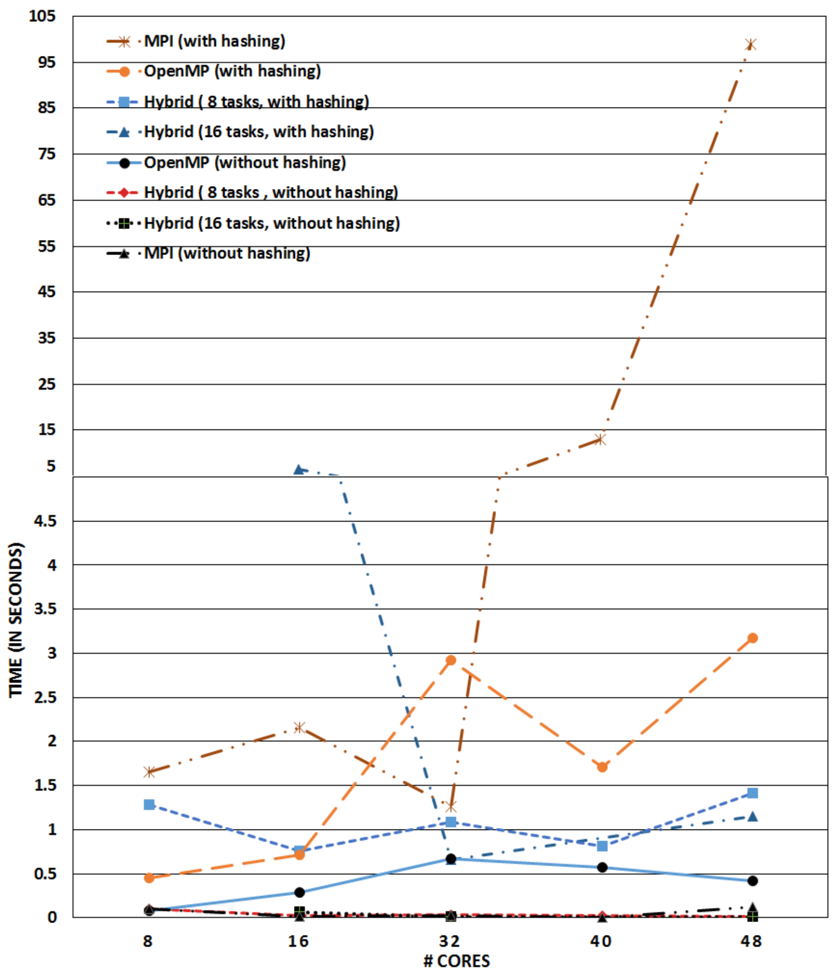


Figure 3.22 CoMD Standard Deviation corresponding to strong scaling

With hashing turned off, the likely source of contention is UDN for MPI-only, UDN+Cache coherence networks (MDN, TDN and CDN) for hybrid, and cache coherence networks for OpenMP-only. With hashing turned on, the dynamics change and the source of contention becomes the cache coherence networks 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 MPI-only) 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, causing more contention on cache coherence networks compared to MDN and UDN alone.

For CoMD, the thread local data always fits in the LLC but the halo data always exceeds the cache. The contention due to off-chip memory accesses always dominates but its impact is reduced as we increase the MPI process count. With hashing turned off, there is additional contention on the UDN (for MPI-only), UDN+cache coherence networks (for hybrid) and cache coherence networks (for OpenMP-only), respectively. With hashing turned on, the input still does not fit into the cache and since almost all accesses are remote, there is a combination of contention on the cache coherence networks and the MDN due to off-chip memory accesses for the latter.

The jitter and average execution times of non-hashed executions remains low in spite of data exceeding the L2 cache capacity compared to hashed execution. In case of non-hashed execution, the likely number of capacity misses increases compared to hashed execution. But the spatial and temporal locality is higher compared to hashed execution. In case of hashed execution, almost all accesses are remote, which increases NoC contention and compromises spatial and temporal locality resulting in longer average execution time and increased jitter.

The analysis of SD and average execution time also concurs with the weak scaling observations in that hybrid execution is most resilient to hashing.

3.6 Related Work

Hung et al. [Hun09] analyze benchmark performance on many-core platforms. They 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. [Ser11] assess the performance and scalability of UPC (Unified Parallel C) over

GASNet based on Pthreads and OperaMPI on the Tile64 many-core architecture. Their work sheds light on various new optimization opportunities due to the NoC architecture of the Tile64 processor along with the challenges involved. They show how one can combine different networks of the Tilera architecture with different properties to improve performance. The drawback of the study is that they analyze Pthreads and MPI separately but the detailed analysis of hybrid OpenMP+MPI is missing.

Suh et al. [Suh12] present a performance analysis of FFT and CRBLASTER on the Maestro processor (derived from the Tilera architecture) and showed that significant speedup can be achieved with the Maestro architecture. Similarly, Singh et al. [Sin11] analyze the performance and scalability of FFTW (Pthreads-based parallelization) and CAF (iLib-based shared memory parallelization). They show that significant speedup can be achieved by using the Maestro many-core architecture. The drawback of their approach is that they consider only shared memory parallelism.

Martin et al. [Mar12] provide an analysis of cache coherence and shared caches within the context of scalability acknowledging the fact that it does not improve scalability of all algorithms. The main drawback of this paper is its limitation to multi-core architectures, i.e., they do not considering the latencies introduced due to a NoC (which is primarily used on modern many-core architectures). It also does not consider the impact of a NUMA architecture.

Jost et al. [Jos03] and Rabenseifner et al. [Rab09] compare various programming models on a cluster with SMP nodes but they do not consider the analysis of the same on many-core architectures which is our main objective.

Chapter 4

Conclusion

In this work, we presented

1. A prototype user-space bank- and controller-aware allocator; and
2. The assessment of programming models comparing MPI-only, OpenMP-only and a hybrid model on the Tilera architecture.

For (1), we found that targeted allocation restricts tiles to access memory of specific banks in addition to a specific controller, thus minimizing bank sharing, balancing and enhancing available bandwidth and performance isolation.

Using this allocator, we performed experiments with a composite benchmark derived from the Mälardalen benchmark suite, the STREAM and the NAS IS OMP benchmarks. Based on our results, we conclude that bank-aware allocation improves performance isolation and available memory bandwidth but NoC contention remains a problem.

One disadvantage of the presented approach is the explicit modification required to use the allocator. If we integrated it into the kernel, it could be seamlessly used by all applications without any modification.

As of now, the allocator supports only shared memory. If we modified the kernel to support bank-aware allocation, it would be possible to support bank-aware allocation across multiple processes as well.

One of the areas of future work is the incorporation of cache partitioning and memory channel partitioning, which (given multiple channels and multiple caches of a high speed NoC) could further improve performance isolation.

Another idea is to add rank awareness to the allocator. This would increase the availability of banks if multiple threads happen to be sharing a given bank similar to prior work [Jeo12].

For (2), we show that programming models affect performance depending on an array of factors. We utilize weak and strong scaling experiments on various benchmarks to understand the impact of NoC latencies on scalability.

Our results indicate that the MPI-like abstraction is highly scalable. But when we induce hashing, we see a significant deterioration in performance for larger numbers of cores due to the stack sharing.

Our results also indicate that shared memory parallelism can offer good scalability but this is not always the case. In case of CoMD, we observed that MPI and Hybrid programming (with higher number of MPI-tasks) models provide good scalability while OpenMP does not. It shows a performance deterioration as we increase the number of cores. But results also show that enabling page hashing by distributing the accesses across cores can sometimes reduce the execution time. Based on the results and analysis of benchmarks, it also shows that CoMD does not take advantage of ccNUMA architectures, thus the benchmark is not ideal for OpenMP-only parallelization. Due to the benchmark design, it is not ideal for hybrid parallelization with lower number of MPI tasks.

Based on the results, we can conclude that the hybrid model, although not the best in terms of scalability, was found to be more robust to NoC latencies and contention compared to the other two programming models as it utilizes the UDN along with other memory management networks. OpenMP-only and MPI-only can offer scalability and performance but are more susceptible to NoC contention (introduced by hashing) as shown by average execution times and SDs for weak scaling and strong scaling.

Based on the analysis of performance models on the Tilera platform, we conclude that NoC latencies and contention are a key consideration when analyzing the scalability on many-core architectures.

Given the results of targeted allocation via our allocator and analysis of performance models, we conclude that our initial hypothesis about performance benefits from targeted allocation and of hybrid programming model is partially true. Even though bank-aware and controller-aware allocation will tend to yield better performance, the same is not the case with the hybrid MPI+OpenMP model. In our case, the Tilera architecture offers a lot of optimization opportunities due to six NoCs, each dedicated for a certain purpose. But not all available manycore architectures offer such opportunities. Each manycore platform may further pose different challenges depending on the architecture. And given this variability, the hybrid model may not be always the best choice.

BIBLIOGRAPHY

[Ada14] Adams, M. “HPGMG 1.0: a benchmark for ranking high performance computing systems” (2014).

[Ada] *Adapteva Processor Family*. `www.adapteva.com/products/silicon-devices/e16g301/`.

[Ake07] Akesson, B. et al. “Predator: a predictable SDRAM memory controller”. *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM. 2007, pp. 251–256.

[Åke07] Åkesson, B. et al. “Real-Time Scheduling of Hybrid Systems using Credit-Controlled Static-Priority Arbitration” (2007).

[Amd67] Amdahl, G. M. “Validity of the single processor approach to achieving large scale computing capabilities”. *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pp. 483–485.

[Tila] *Application Libraries Reference Manual*. `"http://www.tilera.com"`. Tilera.

[Bai91] Bailey, D. H. et al. “The NAS parallel benchmarks”. *International Journal of High Performance Computing Applications* **5**.3 (1991), pp. 63–73.

[Bor07] Borkar, S. “Thousand core chips: a technology perspective”. *Proceedings of the 44th annual Design Automation Conference*. ACM. 2007, pp. 746–749.

[Din11] Ding, X. et al. “SRM-buffer: an OS buffer management technique to prevent last level cache from thrashing in multicores”. *Proceedings of the sixth conference on Computer systems*. ACM. 2011, pp. 243–256.

[Com] *Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). CoMD: A classical molecular dynamics mini-app*. `http://exmatex.github.io/CoMD/doxygen-mpi/index.html`.

[Cos] *Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). CoSP2 Proxy Application*. `http://www.exmatex.org/cosp2.html`.

[Goo13] Goossens, S. et al. “Conservative open-page policy for mixed time-criticality memory controllers”. *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium. 2013, pp. 525–530.

[Gus88] Gustafson, J. L. “Reevaluating Amdahl’s law”. *Communications of the ACM* **31**.5 (1988), pp. 532–533.

[Gus10] Gustafsson, J. et al. "The Mälardalen WCET Benchmarks: Past, Present And Future." *WCET* **15** (2010), pp. 136–146.

[Her11] Herter, J. et al. "CAMA: A predictable cache-aware memory allocator". *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on.* IEEE. 2011, pp. 23–32.

[Hun09] Hung, Y.-F. et al. "Parallel Implementation and Performance Prediction of Object Detection in Videos on the Tilera Many-core Systems". *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on.* IEEE. 2009, pp. 563–567.

[Mic] *Intel Xeon Phi.* `https://www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html`. April,2015.

[Jan13] Jantz, M. R. et al. "A framework for application guidance in virtual memory systems". *ACM SIGPLAN Notices.* Vol. 48. 7. ACM. 2013, pp. 155–166.

[Jeo12] Jeong, M. K. et al. "Balancing DRAM locality and parallelism in shared memory CMP systems". *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on.* IEEE. 2012, pp. 1–12.

[Jos03] Jost, G. et al. "Comparing the OpenMP, MPI, and hybrid programming paradigms on an SMP cluster". *Proceedings of EWOMP.* Vol. 3. 2003, p. 2003.

[Kan09] Kang, M. et al. "Mpi performance analysis and optimization on tile64/maestro". *Proceedings of Workshop on Multi-core Processors for SpaceOpportunities and Challenges Held in conjunction with SMC-IT.* 2009, pp. 19–23.

[Kar13] Karlin, I. et al. "Lulesh 2.0 updates and changes". *Livermore, CA, August* (2013).

[Kim04] Kim, S. et al. "Fair cache sharing and partitioning in a chip multiprocessor architecture". *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques.* IEEE Computer Society. 2004, pp. 111–122.

[Lie97] Liedtke, J. et al. "OS-controlled cache predictability for real-time systems". *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE.* IEEE. 1997, pp. 213–224.

[Lin08] Lin, J. et al. "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems". *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on.* IEEE. 2008, pp. 367–378.

[Liu12] Liu, L. et al. “A software memory partition approach for eliminating bank-level interference in multicore systems”. *Proceedings of the 21st international conference on Parallel architectures and compilation techniques.* ACM. 2012, pp. 367–376.

[Man13] Mancuso, R. et al. “Real-time cache management framework for multi-core architectures”. *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th.* IEEE. 2013, pp. 45–54.

[Mar12] Martin, M. M. et al. “Why on-chip cache coherence is here to stay”. *Communications of the ACM* **55**.7 (2012), pp. 78–89.

[McC95] McCalpin, J. D. “Memory bandwidth and machine balance in current high performance computers” (1995).

[Tilb] *Multicore Development Environment Optimization Guide.* `"http://www.tilera.com"`. Tilera.

[Tilc] *Multicore Development Environment System Programmer’s Guide.* `"http://www.tilera.com/"`. Tilera.

[Mur11] Muralidhara, S. P. et al. “Reducing memory interference in multicore systems via application-aware memory channel partitioning”. *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM. 2011, pp. 374–385.

[Nes07] Nesbit, K. J. et al. “Virtual private caches”. *ACM SIGARCH Computer Architecture News* **35**.2 (2007), pp. 57–68.

[Pao09] Paolieri, M. et al. “An analyzable memory controller for hard real-time CMPs”. *Embedded Systems Letters, IEEE* **1**.4 (2009), pp. 86–90.

[Par13] Park, H. et al. “Regularities considered harmful: forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems”. *ACM SIGPLAN Notices.* Vol. 48. 4. ACM. 2013, pp. 181–192.

[Tild] *Programming The Tile Processor.* `"http://www.tilera.com/"`. Tilera.

[Rab09] Rabenseifner, R. et al. “Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes”. *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on.* IEEE. 2009, pp. 427–436.

[Rei11] Reineke, J. et al. “PRET DRAM controller: Bank privatization for predictability and temporal isolation”. *Proceedings of the seventh IEEE/ACM/IFIP international con-*

ference on Hardware/software codesign and system synthesis. ACM. 2011, pp. 99–108.

[Scca] *SCC External Architecture Specification (EAS) Revision 0.94.*

[Ser11] Serres, O. et al. "Experiences with UPC on TILE-64 processor". *Aerospace Conference, 2011 IEEE.* IEEE. 2011, pp. 1–9.

[Sin11] Singh, K. et al. "Fftw and complex ambiguity function performance on the maestro processor". *Aerospace Conference, 2011 IEEE.* IEEE. 2011, pp. 1–8.

[Sccb] *Single-chip Cloud Computer.* `blogs.intel.com/research/2009/12/sccloudcomp.php`.

[Soa08] Soares, L. et al. "Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer". *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society. 2008, pp. 258–269.

[Suh12] Suh, J. et al. "Implementation of fft and crblaster on the maestro processor". *Aerospace Conference, 2012 IEEE.* IEEE. 2012, pp. 1–6.

[Tile] *Tile Processor I/O Device Guide.* `"http://www.tilera.com"`. Tilera.

[Tilf] *Tile Processor User Architecture Overview.* `"http://www.tilera.com"`. Tilera.

[Tilg] *Tile Processor User Architecture Reference.* `"http://www.tilera.com"`. Tilera.

[Tilh] *Tilera Board Test Kit Guide.* `"http://www.tilera.com"`. Tilera.

[Tili] *Tilera Processor Family.* `www.tilera.com`.

[War13] Ward, B. C. et al. "Outstanding paper award: Making shared caches more predictable on multicore platforms". *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on.* IEEE. 2013, pp. 157–167.

[Wen07a] Wentzlaff, D. et al. "On-chip interconnection architecture of the tile processor". *IEEE micro* 5 (2007), pp. 15–31.

[Wen07b] Wentzlaff, D. et al. "On-Chip Interconnection Architecture of the Tile Processor". *IEEE Micro* **27** (2007), pp. 15–31.

[WJ03] Wijngaart, R. F. Van der & Jin, H. "Nas parallel benchmarks, multi-zone versions". *NASA Ames Research Center, Tech. Rep. NAS-03-010* (2003).

[Wu13] Wu, Z. P. et al. “Worst case analysis of DRAM latency in multi-requestor systems”. *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th.* IEEE. 2013, pp. 372–383.

[WM95] Wulf, W. A. & McKee, S. A. “Hitting the memory wall: implications of the obvious”. *ACM SIGARCH computer architecture news* **23**.1 (1995), pp. 20–24.

[Yun14] Yun, H. et al. “PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms”. *IEEE Real-Time and Embedded Technology and Applications Symposium.* Vol. 356. 2014.

[Zha09] Zhang, X. et al. “Towards practical page coloring-based multicore cache management”. *Proceedings of the 4th ACM European conference on Computer systems.* ACM. 2009, pp. 89–102.

[ZM14] Zimmer, C. & Mueller, F. “NoCMsg: Scalable NoC-Based Message Passing”. *International Symposium on Cluster, Cloud and Grid Computing.* 2014.