Performance effects on HPC workloads of global memory capacity sharing

Rajiv Nishtala
Norwegian University of Science and Technology
rajiv.nishtala@ntnu.no

Paul Carpenter
Barcelona Supercomputing Center
paul.carpenter@bsc.es

Xavier Martorell
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
xavier.martorell@bsc.es

ABSTRACT

While most HPC systems use the traditional "shared nothing" system architecture, with self-contained nodes communicating via the device I/O and associated software layers, there are several ongoing initiatives to build systems that share resources at a coarser granularity, even across the whole machine. Such systems open up new opportunities for resource allocation, in which memory is managed as a shared resource, with an aim to improve time to completion and/or throughput of production workloads. This is expected to be especially beneficial due to the large differences in HPC workloads memory bandwidths and per-node memory footprints.

As a first step towards tackling the resource allocation problem, this paper provides a simulation-based methodology for characterising the effect of memory capacity sharing on performance. The simulation methodology allows characterisation of performance on architectures that are not yet readily available, and it enables co-design of the architecture. To that end, we extend MUSA, a multi-node simulator [13], to simulate an infrastructure that implements inter-node memory capacity sharing. We carried out experiments on real-world HPC workloads: AMG, SPECFEM3D and HYDRO and evaluated the simulations when 50% and 100% of the requests from the remote workload are contending for memory on the local node. Our results show that complementary workloads experience a performance impact of 20 %, respective to their baselines.

1 INTRODUCTION

There is a constant demand for high performance computing (HPC) systems that deliver higher energy efficiency and performance. Although it is important for any new system to rank highly on benchmarks such as High-Performance Linpack (HPL) and High-Performance Conjugate Gradients (HPCG)1, the real challenge is faster time to completion and/or higher throughput of production workloads, as experienced by users. One opportunity to improve such metrics, so far not well captured in benchmarking methodologies, lies in exploiting the large differences in workload memory bandwidths and per-node memory footprints. Application memory footprints, for instance, have been found to range from under 100 MB per node to over 128 GB per node [25], depending on the workload domain, the number of processes, and whether the usage model is strong or weak scaling.

While the dominant approach to building HPC systems is the traditional "shared nothing" system architecture, in which self-contained heavyweight nodes communicate only across device I/O and associated software layers, there are several ongoing efforts to build prototypes that share resources at a coarser granularity, potentially the whole machine. Examples of such initiatives include HP's Machine, dRedBox [2] (with a modular approach based on compute and memory nodes), HPE's Astra supercomputer [24], IBM research [23] and the EUROSERVER, ExaNoDe, and EuroEXA [10] family of projects (whose UNIMEM approach [10] offers the ability to access memory located in remote nodes). These systems open up new opportunities in resource allocation, with a particular impact on the job scheduler (e.g., SLURM [22]), which are given the means to manage memory as a shared resource. Workloads with high memory demands can be submitted to the regular queue, avoiding the need to wait for large memory nodes, of which there are typically only a small number.[1] It becomes possible to request sufficient memory on the regular queue, without having to over-provision compute needs, thereby freeing compute capacity for the other users.

Furthermore, resource allocation simulations on large-scale HPC clusters [21] estimate that the average wait time for a single large memory node is 6 h. As the memory footprint of the job increases, so do the losses in energy due to the over-provisioning of compute nodes (fewer tasks per node) leading to idle, but reserved nodes. These new systems, on the other hand, can influence the job scheduler to allocate resources for a workload and save energy.

As a first step towards tackling the resource allocation problem, this paper provides a methodology for characterising how memory capacity sharing across nodes will impact performance. For instance, depending on a workload's demand for memory bandwidth, it may or may not be significantly impacted by higher latencies to distant memories. In addition, two concurrent workloads may experience contention on a shared interconnect, memory controller or memory bank [7]. Owing to the differences among architectural approaches in this nascent field, and the chicken-and-egg relationship between system architecture on the one hand and efficient resource allocation on the other, we use an approach based on simulation. Our simulation methodology therefore allows characterisation of workloads on architectures that are not yet readily available. This also allows co-design of the architecture, taking account of interconnects, memory systems and workload behaviour. To that end, we extend the MUSA [13]: a multi-node simulator, to simulate an infrastructure that implements inter-node memory capacity sharing.

In summary, the contributions of this paper are: (1) A simulation infrastructure to share global address space across nodes; (2) We carried out experiments using real-world HPC workloads like AMG, SPECFEM3D and HYDRO and provide a quantitative analysis on the performance impact due to memory capacity sharing across nodes; (3) Demonstrate through experimental analysis that global memory sharing has the potential to deliver greater effective performance at a lower cost. Specifically, our results show that when 50 % and 100 % of the requests from the remote workload are contending for memory on the local node, the impact on complementary workloads is < 20 %, respective to their baselines. The characterisation

[1] For instance, MareNostrum-4 at the Barcelona Supercomputing Center has 3,240 regular nodes, with 96 GB (2 GB per core), but only 216 large memory nodes, with 384 GB (8 GB per core).

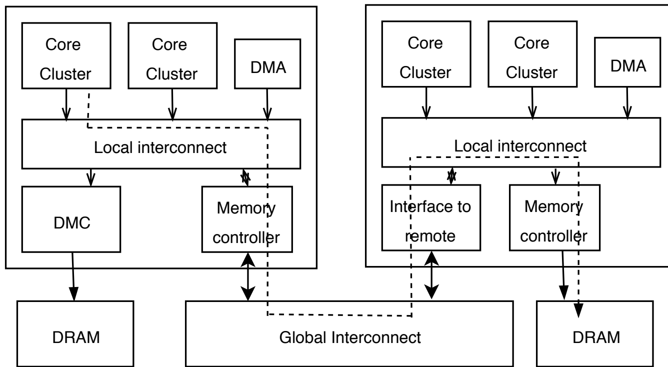

Figure 1: Model architecture based on UNIMEM [10]. The figure shows a remote memory access routed from a node to another.

provides the basis for an intelligent scheduling strategy that can improve time to completion and throughput of workloads, leading to a better user user experience.

The rest of this paper is organised as follows. Section 2 provides a background on the model architecture based on UNIMEM and simulation infrastructure. Section 3 describes the methodology to simulate a shared global address space across nodes. Section 4 presents the benchmarks, the experimental setup and analyses the results. Section 5 discusses the related work. Finally, Section 6 and 7 present the future work and conclusion, respectively.

2 BACKGROUND

This section provides a background on the global shared memory architecture, the simulator infrastructure and the trace collection methodology.

2.1 Global shared memory architecture

State-of-the-art HPC infrastructures have numbers of computing cores on the order of thousands, if not millions, working and communicating in tandem to solve a problem. There are three potential problems that may arise: (a) Big-data and large-scale HPC workloads require large and fast memory subsystems for computations, else suffering from a high disk access latency; (b) These workloads spent a large percentage of the parallel phases in communication, rather than computation, leading to a wastage of actual compute nodes. (c) Workloads with high memory demand have a longer wait time, while the small memory nodes are idle, leading to a bad user experience.

Although NUMAcc machines are available with hundreds of sockets, e.g., SGI Altix [3] or Bull Coherent Switch (BCS) [1], it is difficult or expensive to scale such cache coherent systems beyond a few sockets. Typical HPC systems are therefore built from thousands of NUMAcc nodes, each of which runs an independent OS image, communicating via an interconnect such as InfiniBand. An alternative approach, which we label "global shared memory architecture" is intermediate between the two approaches.

The model architecture is similar to UNIMEM [10], in that it offers the ability to access areas of memory located in the remote nodes at relatively low latency and communication cost. The model architecture achieves this by communicating using load/store instructions and the Remote Direct Memory Access (RDMA) operation through its Global Address Space, which delivers data in-place and avoids receiver-side copying. The "low" latency and communication cost are achieved using the Input/Output Memory Management Unit (IOMMU) and DMA Engine virtualisation, which allows user-level initialisation of RDMA operations. This allows the architecture to facilitate sharing of large memory nodes.

Figure 1 illustrates the model architecture based on UNIMEM. Each node inside the architecture executes an independent OS image, and the hardware provides multiple core clusters, a DMA engine and a local interconnect. These nodes are interconnected using a global interconnect, which permits remote memory accesses.

The global interconnect allows one node to issue remote load or store requests to another. This has two potential impacts: (a) A node can borrow more DRAM from another node. This memory will be accessed and cached only in the borrower's node; (b) A node can share (only) a part of its local DRAM with another node. This memory space will be cached only on the lender node but can be accessed by multiple nodes.

[Exploratory system architecture]: Given the pretext that there exists no production global shared memory architecture, we explore the option of using existing production architectures. What is essential in such an architecture are: (1) Different latencies across sockets to emulate distant memory islands; (2) Cluster of multiple sockets, as HPC workloads typically run in the order of thousands of cores.

Therefore, what is needed is a multi-node simulator to emulate global shared memory architecture to exploit different architectural behaviours and understand the performance effects of memory capacity sharing. This simulation based methodology provides two important advantages (1) it can be applied even when there is no appropriate hardware, and (2) it scales to large numbers of nodes without needing to build a specially configured cluster of the appropriate size.

2.2 Multi-core Simulators

In this paper, we use **MUSA**: a multi-node and multi-level simulation approach [13] that is provided as a plugin to **TaskSim**. TaskSim [20] is a trace-based single node simulator.

[TaskSim]: We simulate the multi-core processor and a DRAM using TaskSim [20] and Ramulator [16], respectively. We employ the Least Recently Used (LRU) page replacement policy in Ramulator. The simulated multi-core processor is an abstract x86 core, with (some) micro-instruction decoding to ensure that issuing or committing of instructions is faster. The dynamic binary translation tool is DynamoRio [9].

[MUSA]: MUSA [13] is a multi-node and multi-level simulation approach that enables different levels of hardware details, simulation cost and accuracy. The multi-core simulator is TaskSim. The network interactions are modelled using Dimemas [5], which can model Message Passing Interface (MPI) communication primitives. MUSA captures inter-node, intra-node and system software communications. MUSA relies on a trace-driven based simulation of different communication networks, the number of cores per node, and other micro-architectural parameters.

3 GLOBAL MEMORY SIMULATION

This section describes the simulation platform and the approach to emulate the global shared memory architecture.

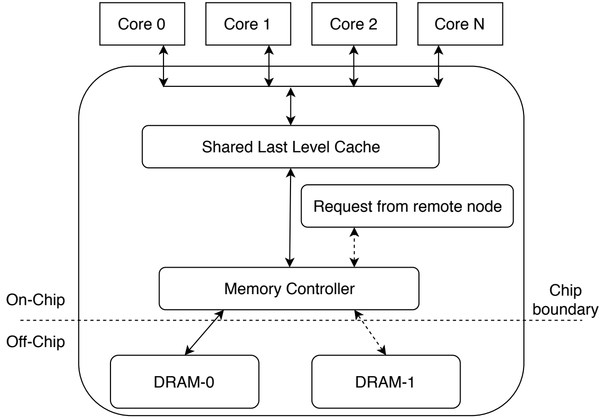

Figure 2: Simulated global shared memory architecture platform. The boldfaced and dotted lines indicate access to the local and remote memory, respectively. The contention model is setup to introduce simulated requests from remote node.

Algorithm 1 Simulated global shared memory architecture

```
1  Let Q be the requests in the pending queue
2  Let X be the total number of DRAM modules
3  Let f[x] be the ratio of requests redirected to DRAM x
4  Let g[x] be the latency to access to DRAM x
5  Let P be the probability at which requests are issued by a remote node
6  Let t be the current clock tick
7  Let n = k = 0                                        ▷ Initialise counters to 0
8  Func WhichDRAM(*counter, *x)
9      if *counter < f[*x] then
10         *counter+=1; return *x;                      ▷ Send to DRAM x
11     else                            ▷ Select next DRAM and reset counter to 1
12         *x < X ? *x+ = 1 : *x = 0; *counter = 0; return *x;
13 Func RedirectRequest(*Q)
               ▷ Pending requests, if any, are sent to DRAM at the "right" time
14     if (SizeOf(*Q) > 0) and (*Q.front().latency <= *t) then
15         send request *Q.front().R to DRAM *Q.front().DRAM_x
16         *Q.pop();
17 repeat
18     if (there exists a request R on this clock tick) then
19         DRAM_x = WhichDRAM(&n, &x)
                      ▷ request R has to be processed on DRAM_x at t + g[x]
20         Add (R, t + g[DRAM_x], DRAM_x) to Q
21     if P > RandomValue(0, 1) then
22         Generate a random read/write request R
23         DRAM_x = WhichDRAM(&k, &x)
24         Add (R, t + g[DRAM_x], DRAM_x) to Q
25     Call RedirectRequest(&Q)
26 until Terminated
```

The simulators, TaskSim/MUSA, define components such as cores, caches, DRAM, etc., as modules. Modules are interconnected using ports. We can configure the number of cores, caches, DRAMs, DRAM size, etc., in the TaskSim/MUSA configuration file. We implement the LRU cache replacement policy for L1 and L2 caches and, for the L3 cache, we implement the slice allocation hash function explained in [17].

To emulate the global shared memory architecture, we make two significant modifications to the existing simulation infrastructure. We remind the reader that the primary objective of this architecture is to allow transparent access to memory located on remote nodes and thereby improve the total system throughput.

[Without Contention]: As a first step, we model NUMAcc accesses without any contention from remote nodes. For this, we instantiate a fixed number of DRAM modules at runtime, which are connected to the memory controller. For each read or write request issued by the core, the memory controller redirects the request to a specified DRAM module in a weighted round-robin fashion. There is a fixed rate at which the requests are being redirected to the local DRAM module or the remote DRAM module. The fixed rate is configured through the TaskSim/MUSA configuration file. The different DRAM modules indicate accesses to local or remote sockets at different latencies.

[With Contention]: As a second step, we model the global shared memory architecture platform. As shown in Figure 1, workloads running on a local node can issue a request to remote nodes and vice-versa. Intuitively, this generates contention on the remote nodes memory controller as requests are being issued from two independent workloads. Therefore, to simulate the scenario mentioned above, we introduce a contention model to the existing simulator. The contention model is a framework that issues/generates a pseudo-random read/write request to the memory controller.

Figure 2 illustrates our contention model on a multi-core architecture. The multi-core architecture has two DRAMs attached to the memory controller. The boldfaced and dotted line indicate accesses to/from the local node (DRAM-0) and remote node (DRAM-1), respectively. Details on the latency to access the DRAM are presented in Section 4.1. The contention model, labelled in the figure as "request from remote node", generates requests with a small but fixed probability distribution and the memory controller redirects the request to either the local or remote nodes DRAM at a fixed rate. The probability distribution ensures simulation of different workload behaviours. The fixed rate indicates the percentage of requests being redirected to the local node from the contention model. The probability of contention and the fixed rate are configured through the TaskSim/MUSA configuration file. For instance, a probability of contention set to 5 % indicates that the remote workload is issuing requests and introducing contention to the memory controller of approximately 5 GB/s. We observe that the contention specified in the configuration file is not strictly enforced and has a maximum discrepancy of 0.27×. This discrepancy is due to the opening and closing of pages in Ramulator, which is employed to save power.

The pseudo-code of the request distribution in a simulated global memory architecture is shown in Algorithm 1. The pseudo-code represents the simulation infrastructure after the shared last-level cache. The pseudo-code checks for two conditions at each clock tick: (1) if there exist any new requests; (2) to ensure that the pending requests, if any, are sent to the specified DRAM at the "right" time. We begin the simulation by setting, in lines 1-7, the number of DRAM modules, the ratio at which the memory controller distributes the requests (for example, $f[x]$ = [3,1] implies that 75 % of the requests are local and 25 % are remote), the latency to access the DRAMs, and the probability at which requests are introduced by the simulated remote workload. Whenever there exists a request on a given clock tick (that is, issued by any core, line 18) or through the contention model (that is, $P > 0$, lines 21-22), we assign the requests a specified DRAM in a weighted round-robin fashion (line 8-12). The weights for the round-robin mechanism are determined by $f[x]$. Next, we append the incoming request *R*, the latency to access the DRAM and the DRAM number *DRAM_x* to the pending queue *Q* (line 20 and 24). At each clock tick, we check if the time is "right" and redirect the request to the specified DRAM

Table 1: System parameters for TaskSim/MUSA

	TaskSim/MUSA
System Configuration	
Core Frequency	2.1 GHz
Core Model	4-issue, out-of-order
Architecture	Simplified CPU model
Memory Subsystem	
Cacheline Size	64 bytes
Private L1 I/D Cache (Latency)	32 KB (4 ns)
Private L2 Cache (Latency)	256 KB (8 ns)
Shared L3 Cache (Latency)	20 MB (28 ns)
DRAM	
Simulator	Ramulator
Standard	DDR4-2400
Capacity	32 GB
Organization	4 channels, 1 rank DDR4 512 MB
Instrumentation Tool	DynamoRio

(lines 13-16). This process is repeated until the termination of the workload.

4 EVALUATION

This section introduces the benchmarks and the hardware resources to analyse the performance impact of memory capacity sharing across nodes. At first, we cross-validate the accuracy, in terms of speed-up, of MUSA against bare-metal. Next, the performance impact in a multi-node case is evaluated in two scenarios: without contention and with contention. For the rest of this section, we instantiate two DRAMs depicting the local and remote DRAM (see Figure 2). The simulation parameters are presented in Table 1.

4.1 Experimentation methodology

[Benchmarks]: We use three scientific real-world workloads [6] implemented in the hybrid MPI + OmpSs programming model: SPECFEM3D, AMG and HYDRO. The input classes selected for the workloads allow the system to run with at least thousand x86 cores. Each workload was run with 16 cores per MPI rank and a total of 256 MPI ranks.

[Hardware resources]: We collect the traces for the workloads running on TaskSim and MUSA on the MareNostrum 4 supercomputer. MareNostrum 4 contains 3456 compute nodes. Each node contains two Intel Xeon Platinum 8160 sockets that together comprise 48 cores operating at 2.1 GHz. Hyperthreading was disabled as in most HPC systems. Xeon Platinum processors are connected to main memory through six channels, and each channel is connected to a single 48 GB DDR4-2666 DIMM. Workloads running on MareNostrum 4 were compiled using GCC 7.1.0, OmpSs 17.12 programming model, Nanos++ 0.11a as the runtime environment and Mercurium 2.0.0 compiler. Mercurium is a source-to-source compiler to translate OmpSs annotation clauses to source code. The Nanos++ is responsible for the internal creation and execution of tasks.

[Memory latency]: We determine the memory access latency for evaluation through exploratory experimentation using the results obtained by running the tool: Intel Memory Latency Checker (Intel MLC) [14] on an Intel Xeon-E5 2670.[2] These experiments were carried out when there is no load on the system. latency provided by the tool include the memory controller, memory channel and all the circuitry between the L3 cache and the DRAM itself.

[2]These experiments were conducted on a stand-alone server as Intel MLC requires root permissions to disable the prefetchers, and could not obtain on a production HPC cluster.

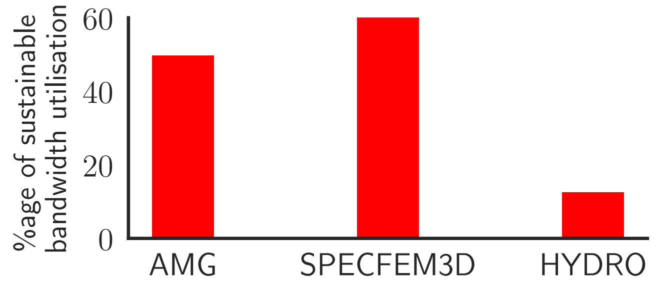

Figure 3: Sustainable bandwidth of the workloads.

Across multiple iterations, the estimated memory latency from the memory controller [4] to the DRAM on the local NUMA socket and remote NUMA socket are 80 ns and 142 ns, respectively. The same latency is used across nodes as we expect them to be physically close on the same board. For the DRAM on the local socket, the access time includes memory device latency and all the components before the memory device. For the DRAM on the remote socket, the access time includes memory device latency, and all components before the global interconnect.

4.2 Results

[Memory bandwidth utilisation]: Figure 3 shows the percentage of sustainable memory bandwidth utilisation for AMG, SPECFEM3D and HYDRO. Based on prior work [15, 19], we set the sustainable bandwidth to 34.9 GB/s, which is 65 % of the maximum theoretical bandwidth. As can be seen, SPECFEM3D and AMG are memory-bound workloads, whereas HYDRO is a compute-bound workload. The simulator takes account of the memory access pattern through the trace. The classification of the results into compute bound and memory bound is only for presentation. Intuitively, workloads with complementary memory bandwidth utilisation should be coupled to make the best use of the global address space available through our architecture and minimise the negative performance impact on workloads, thereby improving time to completion.

We demonstrate the impact on performance when the workload running on the local socket issues NUMAcc access to a remote socket (i.e., the case without contention) and when a workload running on a remote node issues a request to the local node (i.e., the case with contention).

Figure 4 shows the results for three workloads: AMG (first column, subfigures (a) and (d)), SPECFEM3D (second column, subfigures (b) and (e)) and HYDRO (third column, subfigures (c) and (f)). The rows, from top to bottom, correspond to executions without contention (subfigures (a), (b), and (c)) and with contention (subfigures (d), (e), and (f)). Each workload has 256 MPI ranks and 16 cores per MPI rank. For each subfigure without contention, we normalise the execution time (y-axis) with the percentage of remote bandwidth (x-axis) set to 0 i.e., no access to the remote NUMA socket. For each subfigure with contention, we normalise the execution time (y-axis) with the probability of remote bandwidth (x-axis) set to 5 %.

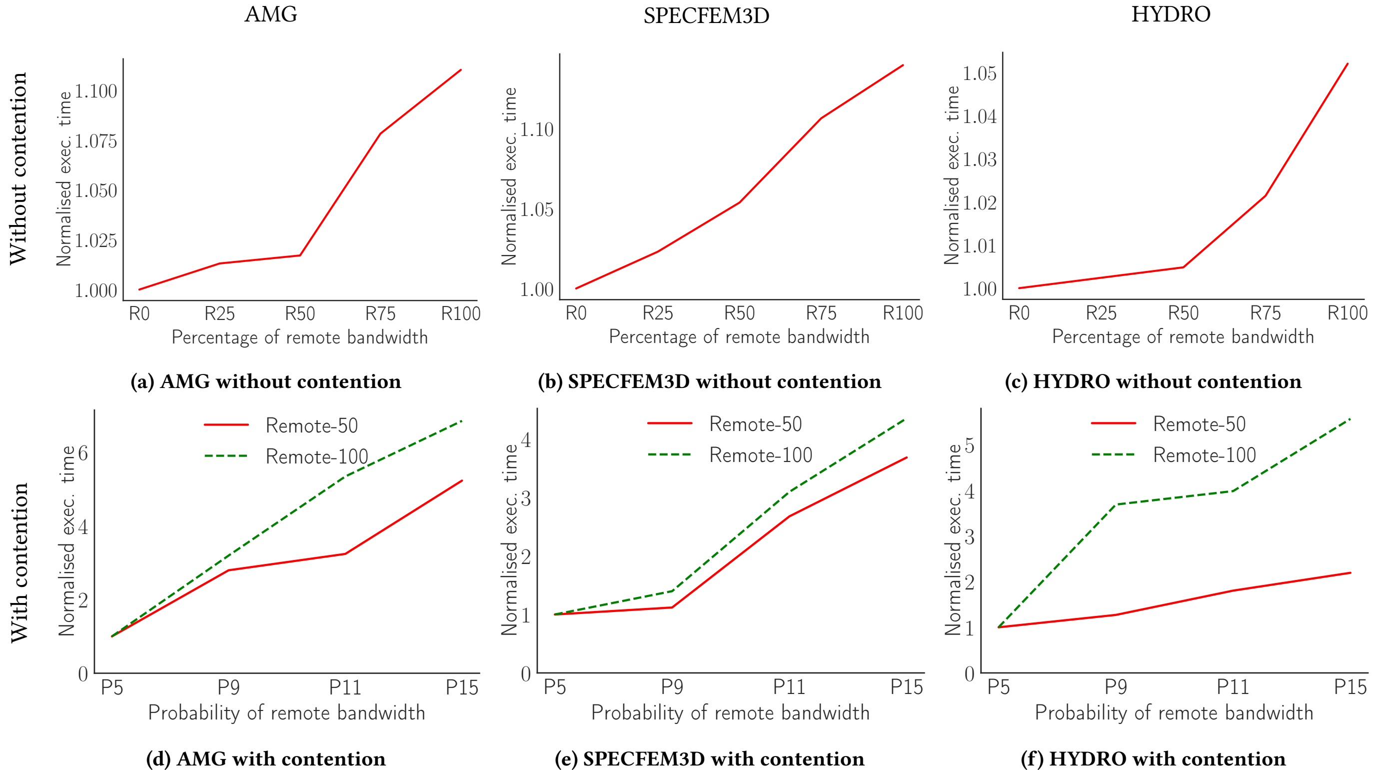

Figure 4: Slowdown of AMG (left-hand column), SPECFEM3D (centre) and HYDRO (right) with without contention (top) and with contention (bottom). Each workload has 256 MPI ranks and 16 cores per MPI rank.

[MUSA without Contention]: Figure 4 (subfigures (a), (b) and (c)) illustrates the performance impact on AMG, SPECFEM3D and HYDRO, respectively when a percentage of remote bandwidth used by the workload is varied. We draw three main conclusions from these graphs: (a) As the percentage of remote memory bandwidth used increases, so does the execution time; (b) For compute-intensive workloads like HYDRO, there is an insignificant slowdown of 1.05×. Also note that for memory intensive workloads like SPECFEM3D and AMG, which generate 3× higher bandwidth when compared to HYDRO, have a slowdown of 1.2×; (c) There is a negligible increase in execution time as none of the workloads cause a performance bottleneck and thus there is a linear relationship between the memory bandwidth and latency [19].

[MUSA with Contention]: Figure 4 (subfigures (d), (e) and (f)) illustrates the performance impact on AMG, SPECFEM3D and HYDRO, respectively, when there is contention from a remote node at a specified probability distribution. In specific, we demonstrate when 50 % and 100 % of the requests issued from the remote node are redirected to the local node. Observe that across all workloads, as the probability of requests issued by a remote node increases, they hit the memory bandwidth "wall" leading to an increase in memory latency and thus, having an exponential slowdown. We draw five main conclusions from these graphs:

(a) For AMG and SPECFEM3D, with the probability of remote bandwidth utilisation set to 15 % ("P15"), we observe that the performance impact is the highest. There are two reasons for this behaviour. (1) AMG and SPECFEM3D are memory bounded workloads and use > 50 % of the sustainable memory bandwidth. (2) Introducing a contention of 15 GB/s on the local nodes' DRAM makes total bandwidth utilisation at approximately 75 GB/s leading to a performance bottleneck. This performance bottleneck increases the bandwidth utilisation of the workload beyond 80 %, where there are more requests introduced at each clock tick than what can be serviced. (b) Across all workloads, we observe that the "Remote-100" has on average 1.6× higher performance impact, in contrast, to "Remote-50", as 100 % of the requests generated by the workload on the local node are redirected to the external node; (c) For SPECFEM3D, comparing "R0" in Figure 4b and "P5" with 100% remote in Figure 4e, we observe that the increase in execution time is as low as 6.53 %; (d) For HYDRO, on the other hand, comparing "R0" (Figure 4c) and "P15" (Figure 4f) with 50 % remote access, we observe that the increase in execution time is 1.24×.

In summary, Figure 7 explores the performance impact in depth, and it shows, for example, that when the workloads have less than 70% of the sustainable bandwidth utilisation, they experience a maximum slowdown of 8.2%. A job scheduler that is able to exploit such opportunities will improve system utilization without imposing a large performance impact. An important conclusion that can, therefore, be drawn from this paper is that global memory sharing has the potential to deliver greater effective performance at a lower cost.

5 RELATED WORK

There have been several research efforts to quantify the performance impact of NUMA access across few hundreds of sockets for single-threaded, big-data and HPC workloads. This work, on the other hand, presents as the basis to quantify the performance impact of global memory capacity sharing. Below we summarise the strongly related research efforts.

[Impact of memory bandwidth on performance]: Diener *et al.* [8] address and analyse that performance of parallel workload is directly impacted by the locality of memory accesses even when executing on cluster systems. Zivanovic *et al.* [25] suggest that large memories represent an attractive solution to significantly reduce costs and energy expenditure in multi-node systems by reducing total number of nodes i.e., scale-in and at the same time increasing the capacity of each node's memory i.e., scale-up. Glew [12] show that a higher bandwidth utilisation may lower the average latency, provided there is sufficient memory level parallelism, which the architecture can exploit. However, a higher bandwidth itself cannot guarantee better performance as it will depend on the inherent MLP in the HPC workloads. Several mechanisms have been proposed in recent years to improve memory locality and thereby the performance of parallel workloads. Dashti *et al.* [7] discuss the impact of memory access imbalance and locality, the main factors that impact performance.

[Resource allocation with job schedulers]: Yiannis *et al.* [11] propose a framework which allows for energy accounting and control of jobs. Patki *et al.* [18] show that power estimations given the job submission time and the flexibility to reduce actual power consumption, can be used to reduce the time from job submission to completion. Contrary to Patki *et al.*, as a part of the future work, we intend to develop a scheduling strategy that takes into consideration the performance impact due to global shared memory and reduce the time from job submission to completion, thereby improving user experience.

6 FUTURE WORK

The goal of the paper is to show the potential of global memory capacity sharing, and to evaluate the performance impact on real workloads. In future work, we plan to build a job scheduler integrated into SLURM. For this, applications will be characterized by signatures (e.g. executable path), which we use at submission time to look up in a database in order to obtain a memory bandwidth bin. Based on this, a heuristic in the job scheduler will schedule workloads with a constraint on the maximum slowdown e.g., $< 10\,\%$.

Specifically, we summarise our future work as follows: (1) The simulation results will be extrapolated to determine the performance impact due to memory capacity sharing on real-world HPC workloads running on multiple nodes using a SLURM simulator; (2) The results obtained show the need for an intelligent scheduling strategy in a job scheduler to share memory resources of compute-bound and memory-bound workloads on an HPC cluster to improve overall system utilisation and to reduce job queue wait-times.

7 CONCLUSION

Modern state-of-the-art HPC systems have traditionally deployed a "shared nothing" system architecture, with self-contained nodes communicating across a high-performance interconnect. In contrast to such architectures, we explore the initiative to share resources among nodes. Specifically, we focused on managing memory as a shared resource and provided a simulation-based methodology to characterise the performance impact on an application as a result of global address space sharing. This is expected to be beneficial due to the substantial differences in HPC workloads memory bandwidths and per-node memory footprints.

Several prior research quantified the performance impact of NUMA access across few hundreds of sockets for single-threaded, big-data and HPC workloads. In contrast to prior efforts, in this paper, we argue that the job scheduler should provide the inter-node memory capacity. We focus on the first and simplest part of the problem, a detailed analysis on the performance impact of global memory capacity sharing on a multi-node simulator (MUSA), to simulate an infrastructure that implements inter-node memory capacity sharing. We quantify the extent to which coupling of two workloads with has an impact on performance, as the contention for bandwidth may or may not degrade the performance. To that end, we carried out experiments using real-world HPC workloads like AMG, SPECFEM3D and HYDRO and show a quantitative analysis when 50 % and 100 % of the requests from the remote workload are contending for memory on the local node. Our results show that complementary workloads experience a performance impact $< 20\,\%$ respective to their baselines. This characterisation provides the basis for an intelligent scheduling strategy that can improve time to completion and throughput of workloads, leading to better user experience.

Acknowledgements – This research is supported by the European Commission under the "Horizon 2020 Framework Programme" with grant number 754337

REFERENCES

[1] 2018. BULL BCS Systems. goo.gl/qR1jML. (2018).
[2] 2018. dredbox: Disaggregated Data Center in a Box. (2018). dredbox.eu/
[3] 2018. SGI Altix UV 1000. goo.gl/1NDCXR. (2018).
[4] Kazi Asifuzzaman and et al. 2016. Performance Impact of a Slower Main Memory: A Case Study of STT-MRAM in HPC *(MEMSYS '16)*.
[5] BSC Application Repository 2018. Dimemas: Predict parallel performance using a single CPU machine. tools.bsc.es/dimemas. (2018).
[6] J Mark Bull and Andrew Emerson. 2018. Selection of a Unified European Application Benchmark Suite. (2018).
[7] Mohammad Dashti and et al. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. *SIGARCH Comput. Archit. News* (2013).
[8] Matthias Diener and et al. 2017. Improving the Memory Access Locality of Hybrid MPI Applications *(EuroMPI '17)*.
[9] DR: DynamoRio 2018. DynamoRio: Dynamic Instrumentation tool Platform. github.com/DynamoRIO/dynamorio. (2018).
[10] Yves Durand and et al. 2014. EUROSERVER: Energy Efficient Node for European Micro-Servers. In *DSD 2014*.
[11] Yiannis Georgiou and et al. 2014. Energy Accounting and Control with SLURM Resource and Job Management System. In *Distributed Computing and Networking*.
[12] Andrew Glew. 1998. MLP yes! ILP no! (1998).
[13] Thomas Grass and et al. 2016. MUSA: A Multi-level Simulation Approach for Next-generation HPC Machines *(SC '16)*.
[14] Intel. 2018. Intel Memory Latency Checker. (2018). software.intel.com/en-us/articles/intelr-memory-latency-checker
[15] Bruce Jacob. 2009. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It.* Morgan and Claypool Publishers.
[16] Yoongu Kim and et al. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE CAL* (2016).
[17] Clémentine Maurice and et al. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters *(RAID 2015)*.
[18] Tapasya Patki and et al. 2015. Practical Resource Management in Power-Constrained, High Performance Computing *(HPDC '15)*.
[19] Milan Radulovic and et al. 2015. Another Trip to the Wall: How Much Will Stacked DRAM Benefit HPC? *(MEMSYS '15)*.
[20] A. Rico and et al. 2011. Trace-driven simulation of multithreaded applications. In *IEEE ISPASS*.
[21] Nikolay A. Simakov et al. 2018. A Slurm Simulator: Implementation and Parametric Analysis. In *HPCS 2018*.
[22] SLURM Workload Manager. 2018. SLURM: Simple Linux Utility for Resource Management. (2018). slurm.schedmd.com/
[23] Dimitris Syrivelis and et al. 2018. A Software-defined SoC Memory Bus Bridge Architecture for Disaggregated Computing *(AISTECS '18)*.
[24] theregister.co.uk 2018. HPE Labs manufactures monster memory Machine system. goo.gl/YVVMip. (2018).
[25] Darko Zivanovic and et al. 2016. Large-Memory Nodes for Energy Efficient High-Performance Computing *(MEMSYS '16)*.