Inception: We need to go wider

Rajiv Nishtala
Barcelona Supercomputing Center
rajiv.nishtala@bsc.es

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

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

I. Introduction

Modern HPC systems are typically built with multiple racks of several multi-core chips put together as a single system. Each such chip has a local DRAM, and they are collectively called as a node. Each node is connected using a high-speed interconnect. This enables the programmer the benefit of transparently issuing a memory request either to local or remote at relative costs.

However, traditional HPC workloads have been shown to have a large variation in their memory footprint, depending on the application domain, the number of processes and whether it is strong or weak scaling. In such circumstances, the programmer is bound to use either the large memory nodes available on commonly deployed HPC systems or suffer large latency delays in disk accesses. For instance, MareNostrum-4 has 128 large memory nodes with 128 GB, as opposed to the typical 32 GB nodes.

On this end, we aim to (1) develop a model to compute the performance overhead of NUMA access latency and application load balancing; (2) reduce the need for large memory nodes by providing HPC resource manager (for example, SLURM) support for memory capacity sharing among nodes using the UNIMEM architecture.

The rest of the paper is organized as follows: Section II provides a background on the related programming models and the UNIMEM architecture. Section III introduces the methodology for validating Inception. Finally, Section IV validates Inceptionusing two different simulators: TaskSim and ZSim .

II. Background

[OmpSs programming model]: The OmpSs programming model is a task-based programming model that provides an abstraction to the implementation of parallel applications. In OmpSs , the task construct enables the annotation of function declaration with the task directive. Every invocation of this function generates a task that is executed concurrently with other tasks or parallel loops. The OmpSs environment is built on top of the (a) Mercurium compiler and (b) the Nanos++ environment which serves as a runtime platform. Mercurium is a source-to-source compiler to translate OmpSs annotation clauses to source code. The Nanos++ is responsible for the internal creating and execution of tasks.

[UNIMEM]: State-of-the-art HPC infrastructures have computing cores in the order of millions, if not billions, working and communicating in tandem to solve a problem. There are two sources for this problem: (1) Big-data applications require large and fast memory subsystems for computations, else suffering from a high disk access latency. (2) These chips spend a lot of energy communicating amongst each other, rather than the actual computation. A large portion of the energy is spent on transferring the communicated data from the remote node buffer to the local nodes' buffer. This, in large, translates to magnitudes of wasted energy and computational overhead.

In contrast to such architectures, the Unified Memory (*UNIMEM*) architecture, offers the ability to access areas of memory located in the remote nodes at a relatively "low" latency and communication cost. UNIMEM achieves this by communicating using the Remote Direct Memory Access (RDMA) operation through its Global Address Space, which delivers data in-place and avoid receiver-side copying. The "low" latency and communication cost is achieved using the Input/Output Memory Management Unit (IOMMU) and DMA Engine Virtualization, which allows user-level initialization of RDMA operations. This allows the UNIMEM architecture to facilitate sharing of large memory nodes.

III. Methodology

[Benchmarks]: We use four scientific workloads implemented in the OmpSs programming model: Blackscholes, dedup, freqmine and fluidanimate. These benchmarks are regularly executed on hundreds of thousands of processing cores. These benchmarks are available as a part of the PARSECSs benchmark suite [5]. For all PARSECSs benchmarks, we use medium and native input datasets [1].

[Hardware resource]: We perform evaluation on two simulators: ZSim and TaskSim . The parameters used in TaskSim and ZSim are presented in Table I.

Nord-III. We collect the traces for the applications running on ZSim on the Nord III cluster [3]. Nord-III contains 84 compute nodes. Each node contains two Intel SandyBridge-EP E5-2670 sockets that comprise of eight cores operating at 2.6 GHz. Hyperthreading was disabled as in most HPC systems. SandyBridge processors are connected to main memory through four channel and each channel is connected to a single 4 GB DDR3-1600 DIMM. Applications running on Nord III were compiled using gcc version 6.2.0, ompss, nanos-0.15a and mercurium-2.1.0.

ZSim. We simulate the multi-core processor and a DRAM using ZSim [13] and DRAMSim2 [12], respectively. ZSim deploys three techniques to achieve accuracy, speed, and scalability. ZSim ensures the accurate x86 code instrumentation

	TaskSim/MUSA	ZSim
Platform	MareNostrum-4	Nord-III
System Configuration		
Core Frequency	3.0 GHz	3.0 GHz
Number of Cores	8	8
Core Model	4-issue, out-of-order	4-issue, out-of-order
Architecture	Simpled CPU model	Intel® Sandy Bridge
Memory Subsystem		
Cacheline Size	64	64
Private L1 I Cache	32 kB 8-way set associative	32 kB 8-way set associative
Private L1 D Cache	32 kB 8-way set associative	32 kB 8-way set associative
Private L2 Cache	256 kB 8-way set associative	256 kB 8-way set associative
Shared L3 Cache	20 MB 20-way set associative	20 MB 20-way set associative
DRAM		
Simulator	Ramulator	DRAMSim2
Standard	DDR3-1600	DDR3-1600
Capacity	8 GB	8 GB
Organization	2 ranks, 8 banks, DDR3 512 MB	2 ranks, 8 banks, DDR3 512 MB
Instrumentation Tool	DynamoRio	Pin

TABLE I: System parameters for TaskSim and ZSim

through dynamic binary translation tool called pin [10]. It speeds up simulation by categorizing memory requests into two-phases: bound and weave phase. Furthermore, it scales well using a user-level OS virtualization layer. The integration of ZSim and DRAMSim2 enables a cycle-accurate simulation for memory requests by creating precise timing events for the weave phase of the ZSim simulator. The simulated multi-core processor is similar to SandyBridge architecture [14].

MareNostrum-4. We collect the traces for the applications running on TaskSim on the MareNostrum-4 supercomputer [2]. MareNostrum-4 contains 3456 compute nodes. Each node contains two Intel Xeon Platinum 8160 sockets that comprise of 48 cores operating at 2.10 GHz. Hyperthreading was disabled as in most HPC systems. Xeon Platnium processors are connected to main memory through six channel and each channel is connected to a single 48 GB DDR4-2666 DIMM. Applications running on MareNostrum-4 were compiled using gcc version 7.1.0, ompss, nanos-0.11a and mercurium-2.1.0.

TaskSim. We simulate the multi-core processor and a DRAM using TaskSim simulator [11] and Ramulator [9]. The simulated multi-core processor is similar to simple CPU model that issues and commits instructions faster. The TaskSim infrastructure uses Nanos++ scheduling delays that happen at the rate of 1 CPU run, and add delays on thread migration and therefore, new task assignment might behave better/worse in some extreme cases. Additionally, TaskSim does not implement a cache coherence protocol, and thereby does not conflict between CPUs. The instrumentation tool used for TaskSim is DynamoRio [7].

[Why Simulator?]: State-of-the-art architectures like Cavium ThunderX [4] provide a dual-socket configuration with large shared memory and high memory bandwidth. In a shared memory system, all processes share a global memory and each processor accesses memory through a shared bus and have a local cache. There is a fixed latency for a memory requests from either socket. The two main concerns with a shared memory system are: contention and coherence. The performance degradation when multiple processors are trying to access the shared memory simultaneously results in contention for memory bandwidth; whereas, having stale data across different cache might result lead to a coherence problem.

Current architectures do not natively provide the possibility to modify the memory access latency or bandwidth from different sockets - as in UNIMEM architectures - and therefore it is hard to emulate multiple memory islands at different latencies.[1]

We emulate the UNIMEM architecture using the aforementioned simulators. These simulators define the core, caches, DRAM, etc., as modules. Every module is connected to one another using "ports". The simulator parameters are configured in the simulators' configuration file. We instantiate a fixed number of DRAM modules at runtime, which are connected to the memory controller. A read or write request from the memory controller is directed to a specified DRAM, at fixed latency, based on the first touch policy [8], which is the default policy in Linux. In the first touch policy, memory is allocated to the same node as the thread that accesses the memory page - this allows to maximize local accesses over remote accesses. However, this is not guaranteed because the data can be shared by threads on multiple nodes.

IV. Evaluation

The aim of this work is two-fold (a) Cross-validating the results obtained from TaskSim (a trace based simulator) and ZSim (an execution driven simulator) with a real-machine. (b) Introduce a simulated contention to the local DRAM from an application from a remote node.

At the time of writing this paper, we are generating the results required.

V. Author biography

Rajiv Nishtala is a Post-doc at the Barcelona Supercomputing Center. His research interests include dynamic resource allocations, energy efficient computing and thread scheduling. For more details: nishtala.github.io

References

[1] C. Bienia and et al. The PARSEC benchmark suite. In *PACT 2008*.
[2] BSC, MareNostrum IV System Architecture.
[3] BSC, Nord III System Architecture.
[4] Cavium. Cavium ThunderX ARM Processor, 2018.
[5] D. Chasapis and et al. PARSECSs. *ACM TACO*, 2015.
[6] H. David and et al. Memory power management via dynamic voltage/frequency scaling. ICAC '11.
[7] DynamoRio: Dynamic Instrumentation tool Platform.
[8] F. Gaud and et al. Challenges of memory management on modern NUMA systems. *Communications of the ACM*, 2015.
[9] Y. Kim and et al. Ramulator: A fast and extensible dram simulator. *IEEE CAL*, Jan 2016.
[10] S. Naftaly. Pin: A Dynamic Binary Instrumentation Tool.
[11] A. Rico and et al. On the simulation of large-scale architectures using multiple application abstraction levels. *ACM TACO*, 8(4):1–20, 1 2012.
[12] P. Rosenfeld and et al. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE CAL*, 2011.
[13] D. Sanchez and et al. ZSim. ACM Press, 2013.
[14] R. S. Verdejo and et al. Microbenchmarks for Detailed Validation and Tuning of Hardware Simulators. In *HPCS 2017*, 2017.

[1] Natively implies without any modifications to the RAS-to-CAS delay and back-to-back CAS delay [6]