Experiments using a Software-Distributed Shared Memory, MPI and 0MQ over Heterogeneous Computing Resources

Loïc Cudennec, Kods Trabelsi

Experiments using a Software-Distributed Shared Memory, MPI and 0MQ over Heterogeneous Computing Resources

Loïc Cudennec[1,2][0000−0002−6476−4574] and Kods Trabelsi[1]

[1] CEA, LIST
F-91191, PC 172, Gif-sur-Yvette, France
`kods.trabelsi@cea.fr`
[2] DGA MI, Department of Artificial Intelligence
BP 7, 35998 Rennes Armées, France
`loic.cudennec@intradef.gouv.fr`

Abstract. Distributed heterogeneous computing systems escalate the problem of choosing the appropriate programming model. Programming models such as message passing are efficient but require low-level management of communications. Higher level of programming such as shared memory are convenient for the application design but they usually have performance issues. With the recent development of distributed heterogeneous systems and new protocols to access remote memories, there is an opportunity for distributed shared memory systems to offer a satisfying level of abstraction while not giving up on performance. In this paper a video processing application is written using MPI, 0MQ and an in-house software-distributed shared memory (S-DSM) backend and deployed over a set of heterogeneous computing boards. Results show that 0MQ implementation is the most efficient but at the price of writing the application with the targeted platform in mind. The S-DSM implementation runs up to 2 times faster than the pure OpenMPI implementation and competes with 0MQ when the data granularity is small.

Keywords: Heterogeneous Computing · Distributed Computing · Distributed Shared Memory · Message Passing.

1 Introduction

Heterogeneous systems are now prevalent in everyday technology including embedded devices, autonomous vehicles, high-performance computing architectures and cloud infrastructures. They offer the possibility to build a specific platform for specific needs in terms of functionality, power processing and energy consumption. However such architectures are complex to program because they escalate the classical problem of hybrid computing in which each resource type exhibits a specific programming interface. Some of these heterogeneous systems are distributed, composed by a mix of heterogeneous computing nodes interconnected by a network, without physical shared memory. For example, microservers are built upon a backplane that provides networking capabilities, power supply and extension slots to host heterogeneous boards such as high-end processors, low-power processors, many-core processors, GPU and FPGA. A common way of programming such platforms is to rely on the message passing paradigm, using popular libraries like MPI and ZeroMQ. With message passing the developer has to manually manage shared data, keep track of their location and initiate the transfers. Another possibility is to use computing frameworks, mainly based on dataflow and workflow programming paradigms such as StarPU. Finally, it is possible to deploy a software-distributed shared memory (S-DSM) that aggregates remote physical memories into a global logical space. The system is in charge of transparently managing shared data and is a step towards single system image (SSI). Using S-DSM allows to conveniently design the application as a regular Posix-like parallel application. S-DSM have been studied from the late eighties with networks of workstations [12], clusters [1], computing grids and clouds [8] and more recently with heterogeneous platforms [6,7]. However it is a common understanding that S-DSM offers poor performances in comparison to message passing, because of the abstraction layer that comes with a price. This explains why S-DSM has never really been used in HPC systems, except for DSM implemented using cache-coherent hardware such as in the Tilera/Mellanox Tile GX many-core processor and further developments in cache coherent interconnects. However these hardware DSM are static by design, usually limited to processors or small homogeneous clusters with dedicated high-performance networks and not prone to be deployed on distributed heterogeneous architectures. Software-DSM are more portable than hardware DSM, they can cope with dynamicity and reconfiguration of the platform and they offer a higher level of abstraction for the application. A few work in the literature evaluate the performance of using a S-DSM compared to message passing. In 1997, Scales and Gharachorloo [17] provide some benchmarks between the Oracle 7.3 distributed database running on 2 DEC AlphaServer 4100 SMP (4 processors) and the Oracle database running on top of the Shasta S-DSM. Results show that using the S-DSM is 2 to 4 times slower than the baseline version. In the late nineties, Bader and Jaja [2] compare the CVM [10] S-DSM to MPICH over the DEC AlphaServer 2100 system, using up to 8 nodes. Results show that MPICH outperforms CVM by a factor of 10. In the early 2000, Werstein et al. [19] evaluate the TreadMarks [1] DSM together with the PVM and MPI message passing frameworks over a Beowulf cluster composed by 32 Intel Pentium III nodes. Results show that the performance of DSM is poorer than PVM and MPI especially when scaling up. However, using the Mandelbrot computing kernel the DSM competes with PVM and MPI. In 2011, Dimakopoulos [18] compares data transfer overheads between MPI and the MOME [8] and MOCHA [11] S-DSM over a 16-node Sun Fire x4100 cluster. Results show that MPI is faster by a factor of 6 to 8 compared to S-DSM. All these results instigate a cold reception whenever a *Yet Another* S-DSM is submitted to the HPC community. Furthermore, S-DSM systems presented in the literature rarely compare the performance of applications running over the S-DSM with the same application running over a message passing framework. With the recent development of high-performance networks, the specification of new remote access protocols such as one-sided communications, RDMA, RoCE, PGAS, OpenCAPI, CCIX, Gen-Z, CXL, there is a renewal of interest in shared memory systems [16,13,5] to unify memory accesses between CPU, GPU, general-purpose accelerators, FPGA and non-volatile memories. Unlike homogeneous computing clusters, such distributed heterogeneous systems require more complex development and tight optimizations to obtain performances. With classical MP-based implementations this complexity is directly exposed to the user. Today S-DSM can play a role not only by offering an abstraction layer, but also by bringing optimization and smart decision for data management directly in the runtime. In the Grappa [14] S-DSM proposed in 2015 the authors show that porting over the S-DSM several computing frameworks such as MapReduce and GraphLab can run up to 1.33 faster than the baseline implementations on a 128-node AMD Interlagos cluster using a Mellanox Infiniband interconnect. In this momentum of renewal, the Argo [9] DSM proposes new coherence mechanisms that allow to match or exceed MPI implementations of some SPLASH and NAS benchmarks running onto a cluster of 128 AMD Opteron NUMA nodes. Note that Argo is implemented on top of MPI to manage remote connections. These are promising results, being the demonstration that a S-DSM can perform better than other MP-based implementations. It also advocates for the use of high-level programming models without giving up on performance. The main contribution of this paper is to report on the ins and outs of writing an application using message-passing and S-DSM. We start from an application specification and we elaborate different implementations to compare performance over a distributed heterogeneous computing platform. These implementations include the well-established message-passing OpenMPI runtime, the lightweight ZeroMQ (0MQ) message-passing runtime and an in-house S-DSM [3,4] built upon OpenMPI and designed to study data management over heterogeneous architectures. Results show that the S-DSM implementation outperforms the pure OpenMPI implementation by a factor 2 and get close to the ZeroMQ implementation performance for data-sets with smaller granularity.

2 Implementations of a Video Processing Application

In this work we consider a video processing application that has been used to experiment and showcase several distributed heterogeneous computing platforms such as the Christmann RECS|Box microserver [15]. This application opens a video stream, either from a file or a camera, decodes the frames, distributes the frames to remote processing tasks and encodes the processed frames back to a file or a live display. The computing kernel is a 3x3 convolution used for edge detection. From this specification we have implemented three versions based on MPI, ZeroMQ and the S-DSM. These versions share the exact same code in `C`, except for data management. Figure 1 illustrates the communication sequences between the input task, the processing tasks and the output task for the different implementations. MPI and ZeroMQ implementations are quite straightforward and

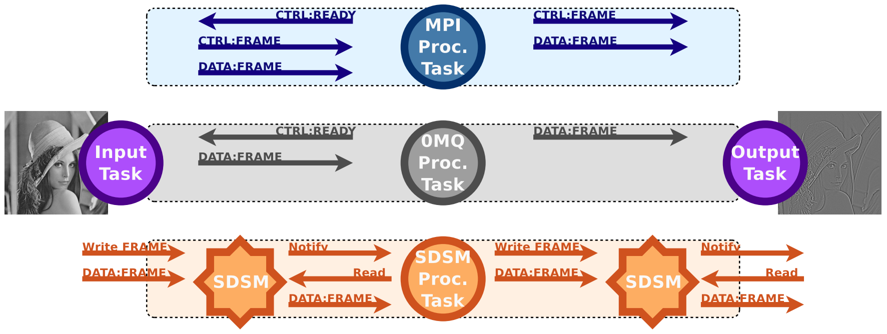

Fig. 1: Transferring and accessing frames in MPI, 0MQ and S-DSM.

are similar to a *split-join* dataflow, using multiple producer-consumer patterns. The S-DSM implementation is more complex (inner communications occurring between S-DSM servers are not represented here) because each access to a shared data triggers multiple communications between S-DSM servers and applications tasks, according to the data coherence protocol. The MPI implementation generates around twice as much messages as ZeroMQ, and the S-DSM generates 10 times more messages than ZeroMQ.

MPI. The global behavior of the MPI implementation is as follows: 1) the processing task sends a control message to the input task to indicate they are ready to take a job, 2) the input task sends a control message with job information followed by the input frame to the processing task and 3) once the frame has been processed, the processing task sends a control message followed by the processed frame to the output task. When deploying more than one processing task, it implements an eager scheduling in which tasks that run faster are whiling to process more frames than the others. This implementation is based on simple MPI concepts including synchronous and asynchronous version of `Send` for sending messages, `Probe` and `Wait` for checking if a message is available and a communication is completed and `Recv` to receive a message. We do not use collective primitives nor advanced group communication operations. There are three variants of the code: 1) *Synchronous single buffer* means that a single buffer is used on the input task to send frames to the processing tasks. *Synchronous* means that the input process waits for the completion of the `Send` operation before decoding and sending the next frame. 2) *Asynchronous single buffer* allows the input task and the processing tasks to not wait for the completion of the `Send` operation, allowing local parallelism between the user code and the MPI runtime (eg. decode next frame, process next frame while sending the previous one). 3) *Asynchronous multiple buffers* means that one buffer is allocated on the input task per processing task, allowing to drastically increase the parallelism between frame decoding and the management of communications in the MPI runtime. In these experiments we use the OpenMPI 3.x runtime because of its popularity and the possibility to compile the source code without a glitch onto different Linux distributions (Ubuntu, Debian, Raspbian, Lebian) and processors (Intel Core i7, Arm Cortex) deployed in our heterogeneous platform.

ZeroMQ (0MQ). ZeroMQ is a lightweight message passing framework released around 2010. It offers a low-latency implementation of sockets based on communication patterns instead of basic message passing. There is no logical process overlay built on top of the communication sockets such as *communicators* and *ranks* for MPI. Therefore, when connecting to a distant node, the IP address or hostname must be known, which is platform-dependent and less elegant. ZeroMQ is expected to be more efficient than MPI notably because there is no node bootstrapping, peer discovery and group communication overlay management. In this implementation of the video processing application, a request-reply *REQ-REP* communication pattern is used between the input task (acting as a server, *REP*) and the processing tasks (acting as clients, *REQ*). The resulting interaction makes the processing tasks ask the input task for the next frame to compute. As for the MPI implementation, it implements an eager scheduling of frames onto processing tasks. The *PUSH-PULL* communication pattern is used between the processing tasks (*PUSH*) and the output task (*PULL*) in order to collect the results. Note that this pattern implements *fair-queuing* which explains it cannot be used between input and processing tasks because it would evenly distribute frames onto processing tasks, hence not implementing an eager scheduling.

S-DSM. The shared memory implementation is based on the S-DSM presented in 2017 in Cudennec [3]. This S-DSM relies on the OpenMPI 3.x runtime in order to manage the underlying peer network and message delivery. It is organized as a super-peer topology made of a peer-to-peer network of S-DSM servers for cache and metadata management, and a set of clients to run the user code. The coherence protocol is a 4-state (MESI) home-based protocol. Shared data are stored into atomic pieces of data called *chunks*. The S-DSM provides a regular interface for accessing chunks and performing distributed synchronizations. It also introduces an event-based programming language in which it is possible to subscribe to chunks in order to be notified whenever the chunk has been modified, as in a publish-subscribe communication pattern [4]. In this implementation of the video processing application, frames are stored in the shared memory: for each processing task a shared input buffer and a shared output buffer are allocated in the S-DSM. The input task writes incoming frames into the input buffer of a ready task. The processing task gets notified, reads the frame from its input buffer and write the processed frame into its output buffer. The output task gets notified that a new result is available, it reads the frame and checks for frame reordering before sending to the output. The resulting application layout is close to a dataflow, which is indeed a common way of implementing dataflow runtimes over shared memory.

Node	Processor	Cores	RAM	Storage	Network	#
Gateway	Intel Core i7 6800K	6	64GB	SSD	Gb Ethernet	1
Raspberry Pi 3B+	ARM Cortex A53	4	1GB	SD	USB 2.0	2
Odroid XU4	ARM Cortex A15/A7	4/4	2GB	SD	USB 3.0	1
Odroid XU3	ARM Cortex A15/A7	4/4	2GB	SD	USB 2.0	1
HiKey Kirin 970	ARM Cortex A73/A53	4/4	6GB	UFS	USB 3.0	2
Nvidia Jetson TX2	Denver/ARM Cortex A57	2/4	8GB	eMMC	Gb Ethernet	1
Adapteva Parallella	ARM Cortex A9/Epiphany	2/16	1GB	SD	Gb Ethernet	0

Table 1: Platform description and number of nodes (#) used in the experiments.

3 Results

The hardware platform is a small cluster of heterogeneous computers and development boards connected to a Gigabit Ethernet switch. Table 1 describes the node types of the platform and the number of nodes that are used in the following experiments. These nodes are representative of the hardware that can be integrated within HPC microservers, cloud infrastructures and platforms for autonomous vehicles, albeit the form-factor of the resulting setup and the poor network performance for some of the nodes connected via Ethernet over USB. In all experiments, a processing task is deployed on each node and the two input and output tasks are co-located on the Core i7 Gateway node. In the specific case of a S-DSM deployment, a S-DSM data server is deployed on the Core i7 node. When deploying the configuration with 4 servers, 3 additional servers are deployed on the Nvidia TX2 and the two Kirin 970 nodes.

Ideal computation time. It is possible to evaluate the *ideal* computation time of the application by measuring the time it takes to run the computational kernel on each node. This information is used to calculate the contribution of each node in the global computation, without considering network communications and other input-output operations. Table 2 presents the processing times measured on each node type to run a convolution (stencil 3x3). The frame size follows the HD, UHD-1 and UHD-2 standards and the corresponding frame representation sizes are 2MB, 8MB and 33MB (as for a 256 bits, greyscale frame). Processing times do not include input and output operations on local storage to read and write the frame. The *ideal* computation time of the application when running on the whole platform can be calculated because the convolution kernel is not data-dependent, which means that its complexity does not depend on the input data, therefore making the convolution processing deterministic. Table 2 presents the results step-by-step. The first step is to calculate the normalized performance of the node, taking the Raspberry Pi 3B+ as reference (RPI performance is set to 1). For example, the normalized performance of the Core i7 for UHD-2 indicates that the Core i7 computes more than 13 times faster than the RPI. The second step is to calculate the workload coefficient per input data-set. This can be done using the following equation (note that we do not use the Adapteva Parallella

		Core i7	TX2	XU3	XU4	Kirin 970	RPI 3B+	Deviation
HD	Time per frame (s)	0.041	0.131	0.145	0.219	0.153	0.341	
	Normalized to RPI	8.317	2.603	2.351	1.557	2.228	1	
	Ideal (nb of frames)	506	158	143	95	136	61	0
	MPI (nb of frames)	178	180	131	152	175	153	680
	S-DSM (nb of frames)	490	226	65	92	119	92	262
	0-MQ (nb of frames)	196	166	147	151	160	158	620
UHD-1	Time per frame (s)	0.103	0.353	0.597	0.864	0.463	1.202	
	Normalized to RPI	11.669	3.405	2.013	1.391	2.596	1	
	Ideal nb of frames	590	172	102	70	131	51	0
	MPI (nb of frames)	173	179	127	158	172	159	834
	S-DSM (nb of frames)	475	228	66	94	124	93	330
	0-MQ (nb of frames)	196	166	146	157	160	157	800
UHD-2	Time per frame (s)	0.342	1.363	2.091	3.478	4.260	4.625	
	Normalized to RPI	13.523	3.393	2.211	1.329	1.085	1	
	Ideal nb of frames	717	180	117	70	58	53	0
	MPI (nb of frames)	355	237	118	118	120	119	724
	S-DSM (nb of frames)	453	234	70	98	126	99	622
	0-MQ (nb of frames)	752	123	40	79	99	58	268

Table 2: Calculating the *ideal* number of processed frames per node type using the Pthread implementation (theory, no communications). Note that we use two RPI and two Kirin boards in our experiments, which is taken into account when calculating the *ideal* number of processed frames. The effective number of processed frames observed in the experiments are given for MPI, S-DSM and 0MQ for each node type. *Deviation* is the cumulative distance with the *ideal* number of processed frames (smaller is better).

board and that there are two RPI and two Kirin boards in the setup):

$$\alpha * (P_{i7} + P_{TX2} + P_{XU3} + P_{XU4} + 2 * P_{Kirin} + 2 * P_{RPI}) = NB\_FRAMES$$

With P_n the normalized performance of node type n, NB_FRAMES the number of frames in the input video and α the unknown workload coefficient. In the following experiments, the HD, UHD-1 and UHD-2 video samples are taken from the '3DMark Port Royal Demo' benchmark, with a total of 1296 frames for HD, 1298 for UHD-1 and 1306 for UHD-2. The α workload coefficient is therefore 60.9 for HD, 50.6 for UHD-1 and 53.0 for UHD-2. The last step is to calculate the *ideal* number of processed frames per node type using the following formula: $NB\_Frames_n = \alpha * P_n$. From this result it is possible to calculate the *ideal* global processing time using the following formula: $GLOBAL\_TIME = max(NB\_Frames_n * TIME\_PER\_FRAME_n)$. This gives $20.8s$ for HD, $61.3s$ for UHD-1 and $247.0s$ for UHD-2. This *ideal* processing time does not include overheads such as distributing the computation over a network, managing the communication buffers and processor caches.

Comparing implementations performance. The three main implementations of the video processing application (OpenMPI 3.x, S-DSM and ZeroMQ

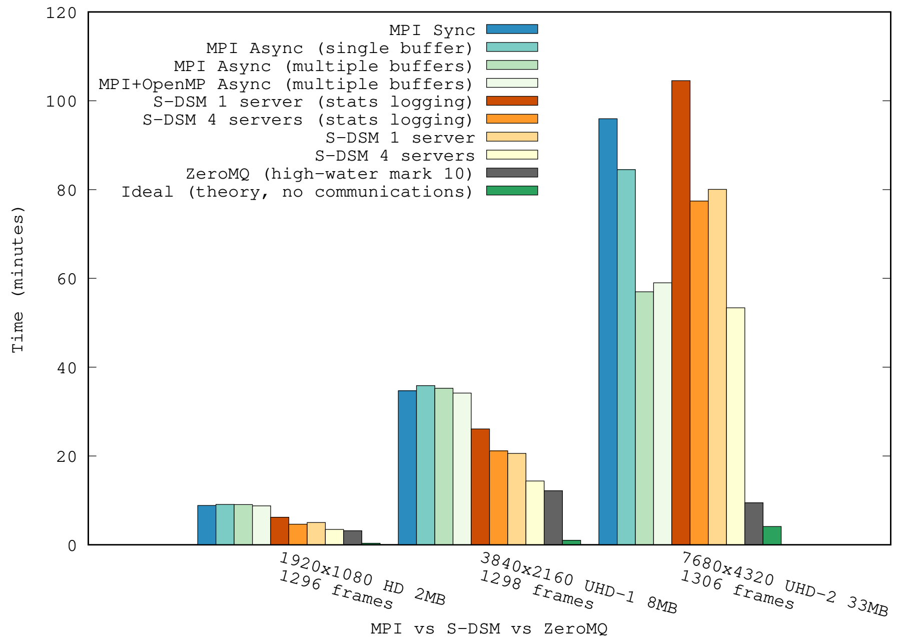

Fig. 2: Comparing the different application implementations.

4.x) have been deployed on the heterogeneous platform and evaluated using the three video samples (HD, UHD-1 and UHD-2). Results are given in Figure 2 and can be compared to the *ideal* computation time. The S-DSM implementation generates around 39000 messages at the MPI level, the MPI implementation generates around 6500 messages and the ZeroMQ implementation generates around 3900 messages. The ZeroMQ implementation is the fastest for each data-set, and even more when dealing with bigger data: the global computing time is smaller when processing UHD-2 frames (568 seconds) than UHD-1 frames (731 seconds). While being a counter-intuitive result, it is usually explained by the adequacy between data granularity and the management of network and processor caches. By default, the ZeroMQ runtime sets the capacity of communication pipes, also called *High-Water Mark*, to 1000 messages (or even no limit for early versions of the runtime) which leads to memory overflow and a `segfault` on nodes with limited physical memory such as the Raspberry Pi. In these experiments, the *High-Water Mark* is set to 10 which allows the proper termination of the application with all data-sets. It also prevents from cache pollution that acts as a performance killer on several nodes. The latter point being one of the main reason why ZeroMQ performs significantly better than the other MPI-based implementations. A second counter-intuitive result is that the S-DSM implementation (over MPI) is performing better than the regular MPI implementation. For HD and UHD-1 it even gives results close to the ZeroMQ implementation. Four configurations of the S-DSM are used, as a combination of enabling or not the logging of events (stats logging) and deploying a single or 4 metadata and cache servers. Stats logging generates around 240000 events per run that are stored in the physical memories of the nodes before being dumped into files at the end of the computation. This implies a significant overhead, while mandatory to finely analyze the S-DSM behavior. Using 4 metadata and cache servers let the S-DSM system balance access requests to shared data among different nodes, hence being more responsive. One of the main reason the S-DSM performs better than MPI is because of the parallelism of data it introduces, similar to a pipeline: each time a shared data is modified, it is sent to the S-DSM servers, and not to the input buffer of another processing task. Therefore, processing nodes do not have to undergo all incoming data, but rather ask for them in a on-demand basis thanks to the S-DSM programming model. For small data-sets (HD and UHD-1), the best MPI implementation computation time is more than 2 times slower than the best S-DSM configuration computation time. For UHD-2, the *async multiple buffers* implementations performances are close to the S-DSM, revealing the importance of manually managing the communication buffers to increase the parallelism degree. However, it is very far from the ZeroMQ performance, which is quite a surprise as it relies on the same programming model.

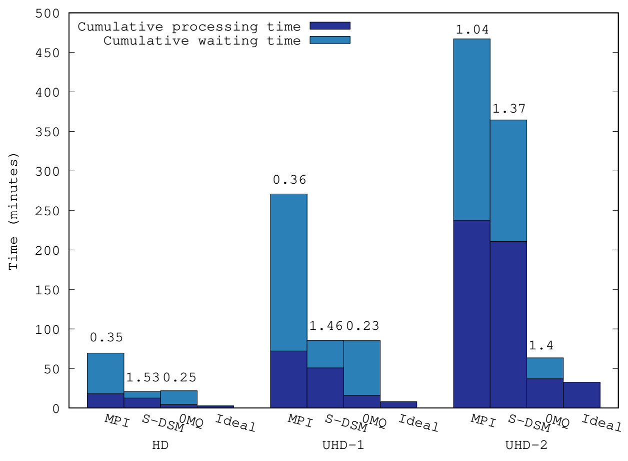

Fig. 3: Cumulative processing and waiting times using the best implementations.

Influence on user code. The user code can be split into two parts: the processing time which is the time spent in the computing kernel, and the waiting time which is the time spent between the end of a kernel call and the beginning of the next call. This waiting time includes operations to asynchronously send the previously processed frame to the output task and to synchronously retrieve the next incoming frame. This is a relevant indicator to know if frames are delivered just in time and to identify a data starvation crisis. Figure 3 presents the cumulative processing and waiting times for the best MPI, S-DSM and ZeroMQ implementations. Labels on top of bars represent processing times divided by waiting times. Beware of this representation: the global processing time of the application as shown in figure 2 is the consequence of particular intricacies of individual processing and waiting times, and cannot be compared to a simple sum of processing and waiting times. There are several conclusions based on this figure. First, the cumulative processing time of ZeroMQ is close to the *ideal* cumulative processing time which means that the frames have been wisely dispatched to the computing nodes and that the ZeroMQ runtime does not interfere with the computing capabilities. Second, the S-DSM performance is close to ZeroMQ for HD and UHD-1 data-sets despite a higher cumulative processing time. As a counterpart, the cumulative waiting time is smaller which indicates that the S-DSM runtime was able to deliver data in a more efficient way than ZeroMQ. In that case, increasing the size of communication pipes (*High-Water Mark*, HWM) for ZeroMQ might decrease the cumulative waiting time but at the price of increasing the memory footprint, degrading the processing performance and even getting a memory overflow as discussed previously. Therefore, there is a trade-off to find when setting an arbitrary value for HWM, which is not acceptable for a regular user. Finally, the pure MPI implementation reveals important processing and waiting times with the three data-sets.

Load balancing. One of the main reason the cumulative processing times and waiting times are increasing comes from a poor load balancing of frames onto computing resources. The three implementations are all based on eager scheduling of frames onto computing resources. Therefore, the effective load balancing of frames is a direct consequence of the underlying communications and data management runtime. It is possible to compare the effective scheduling of frames in the experiments with the *ideal* number of processed frames as presented in Table 2. The Core i7 node is the most powerful node and should process more frames than the other nodes. However in all the experiments the Core i7 node is far from processing the expected number of frames. The cumulative distance from the *ideal* number of processed frames shows that for the smaller data-sets (HD and UHD-1) the S-DSM is able to manage a better load balancing than MPI and ZeroMQ while for a larger data-set (UHD-2) ZeroMQ offers the best load balancing which finally explains why processing UHD-2 is faster than UHD-1. Communication runtimes such as MPI and ZeroMQ are complex distributed software. The inability to achieve a proper load balancing for the smaller data-sets might be the consequence of smart mechanisms against message delivery starvation, which leads to a fair distribution of frames among the nodes instead of favoring the Core i7 node as expected. Note that this underlying behavior is hidden to the application developer. Despite being implemented over MPI, the S-DSM has better load balancing, which can be explained by the important mix of control and data messages exchanged between several nodes whenever accessing a frame in the shared memory. Therefore the communication pattern to access a frame is more complex and more resilient to specific runtime arbitration.

4 Conclusion

Software-distributed shared memory adoption in high-performance computing systems is conditioned upon reaching acceptable performances. In this work, a distributed application has been written over message passing and S-DSM frameworks and deployed over an heterogeneous platform. Results show that the S-DSM implementation is faster than the pure MPI implementation and competes with the lightweight ZeroMQ implementation for small granularity data sets. It appears that the MPI runtime is designed and optimized for supercomputing architectures with strong assumption on hardware capabilities (processor speed, amount of physical memory and networking performance). With the development of distributed heterogeneous architectures, these assumptions are not reliable, especially with low-power processors and embedded devices. In such a context, the S-DSM is able to introduce intermediate storage places which prevents from the overload of communication buffers on processing nodes. The management of such intermediate storage places is transparent for the application, which is inherent to the S-DSM approach compared to message passing frameworks. Several conclusions come with this work: 1) lightweight message passing (0MQ) is faster but at the price of specializing the application to the platform, 2) the OpenMPI runtime is not optimized for running onto low-power processing boards, and 3) S-DSM overhead is getting smaller compared to message passing, as the hardware is becoming more complex to deal with. Therefore, while probably still not being fully adapted to large-scale homogeneous clusters, this work shows that S-DSM is a serious contender to leverage the computing capabilities of distributed heterogeneous architectures and their applications.

Acknowledgments. This work has received funding from the European Union's Horizon 2020 research and innovation action under grant agreement No 688201.

References

1. Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.: TreadMarks: Shared memory computing on networks of workstations. IEEE Computer **29**(2), 18–28 (Feb 1996)
2. Bader, D., Jaja, J.: Simple: A methodology for programming high-performance algorithms on clusters of symmetric multiprocessors (smps). Journal of Parallel and Distributed Computing **58**, 92–108 (07 1999)
3. Cudennec, L.: Software-distributed shared memory over heterogeneous micro-server architecture. In: Euro-Par 2017: Parallel Processing Workshops. pp. 366–377. Springer International Publishing (2018)
4. Cudennec, L.: Merging the publish-subscribe pattern with the shared memory paradigm. In: Euro-Par 2018: Parallel Processing Workshops. pp. 469–480. Springer International Publishing, Cham (2019)
5. Dragojevic, A., Narayanan, D., Hodson, O., Castro, M.: FaRM: Fast remote memory. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation. pp. 401–414 (2014)

6. Gelado, I., Stone, J.E., Cabezas, J., Patel, S., Navarro, N., Hwu, W.m.W.: An asymmetric distributed shared memory model for heterogeneous parallel systems. In: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems. pp. 347–358. ASPLOS XV, ACM, New York, NY, USA (2010)
7. Ghane, M., Chandrasekaran, S., Cheung, M.S.: Towards a portable hierarchical view of distributed shared memory systems: Challenges and solutions. In: Proceedings of the 11th International Workshop on Programming Models and Applications for Multicores and Manycores. PMAM'20 (02 2020)
8. Jegou, Y.: Implementation of page management in MOME, a user-level dsm. In: CCGrid 2003, 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid. pp. 479–486 (05 2003)
9. Kaxiras, S., Klaftenegger, D., Norgren, M., Ros, A., Sagonas, K.: Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing. pp. 3–14 (2015)
10. Keleher, P.: CVM: The coherent virtual machine **TR93-215** (01 1995)
11. Kise, K., Katagiri, T., Honda, H., Yuba, T.: Evaluation of the acknowledgment reduction in a software-dsm system. In: Proceedings of the 6th International Conference on parallel Processing and Applied Mathematics. pp. 17–25 (2005)
12. Li, K.: IVY: a shared virtual memory system for parallel computing. In: Proc. 1988 Intl. Conf. on Parallel Processing. pp. 94–101. University Park, PA, USA (Aug 1988)
13. Mitchell, C., Geng, Y., Li, J.: Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In: Proceedings of the 2013 USENIX Conference on Annual Technical Conference. pp. 103–114 (2013)
14. Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S., Oskin, M.: Latency-tolerant software distributed shared memory. In: 2015 USENIX Annual Technical Conference (USENIX ATC 15). pp. 291–305. USENIX Association, Santa Clara, CA (2015)
15. Oleksiak, A., Kierzynka, M., Agosta, G., Barenghi, A., Brandolese, C., Pelosi, W.F.G., Cecowski, M., Plestenjak, R., Cinkelj, J., Porrmann, M., Hagemeyer, J., Griessl, R., Lachmair, J., Peykanu, M., Tigges, L., v. d. Berge, M., Christmann, W., Krupop, S., Carbon, A., Cudennec, L., Goubier, T., Philippe, J.M., Rosinger, S., Schlitt, D., Adeniyi-Jones, C.P.C., Setoain, J., Ceva, L., Janssen, U.: M2DC - modular microserver datacentre with heterogeneous hardware. Microprocessors and Microsystems **52**, 117–130 (2017)
16. Ross, J.A., Richie, D.A.: Implementing openshmem for the adapteva epiphany risc array processor. Procedia Computer Science **80**, 2353 – 2356 (2016), international Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA
17. Scales, D.J., Gharachorloo, K.: Towards transparent and efficient software distributed shared memory. ACM SIGOPS Operating Systems Review **31**(5), 157–169 (10 1997)
18. V. Dimakopoulos, P.H.: HOMPI: A hybrid programming framework for expressing and deploying task-based parallelism. pp. 14–26 (08 2011)
19. Werstein, P., Pethick, M., Huang, Z.: A performance comparison of dsm, pvm and mpi. pp. 476–482 (09 2003)