A High-Performance Superpipeline Protocol for InfiniBand

Alexandre Denis

A High Performance Superpipeline Protocol for InfiniBand

Alexandre Denis

INRIA Bordeaux Sud-Ouest / LaBRI, France
Alexandre.Denis@inria.fr

Abstract. InfiniBand high performance networks require that the buffers used for sending or receiving data are registered. Since memory registration is an expensive operation, some communication libraries use caching (rcache) to amortize its cost, and copy data into pre-registered buffers for small messages. In this paper, we present a software protocol for InfiniBand that always uses a memory copy, and amortizes the cost of this copy with a superpipeline to overlap the memory copy and the RDMA. We propose a performance model of our protocol to study its behavior and optimize its parameters. We have implemented our protocol in the NewMadeleine communication library. The results of MPI benchmarks show a significant improvement in cache-unfriendly applications that do not reuse the same memory blocks all over the time, without degradation for cache-friendly applications.

1 Introduction

InfiniBand networks are nowadays the leading technology for high performance networks in clusters. Parallel applications usually exploit this network through an Mpi library that makes their usage seamless for the end-user. Under the hood the Mpi implementations access the InfiniBand network cards through an Api called *verbs*. Unlike Api used to program other networking technologies, the *verbs* Api is very low-level. It means that a lot of things have to be done by hand by the Mpi library programmer; on another hand, the programmer has a lot of control on how to exploit the network interface.

Network transfers are based on RDMA and are executed by the DMA engine on the network card. The card sees the system from the PCIe bus, thus works with physical addresses. The application, Mpi library, and InfiniBand software stack run in user space, with no system call involved thanks to OS bypass. Since they run in user space, they use virtual addresses. Thus, when sending data from user space through InfiniBand, translation has to be done from virtual address space to physical address space. The network card can do the translation if it has been told previously the mapping from virtual to physical space. This process is called *memory registration* and in InfiniBand it has to be performed explicitly by the user. Actually, the memory registration is comprised of both the communication of the translation table to the network card, and memory pinning to prevent swapping. All memory involved in sending and receiving operations

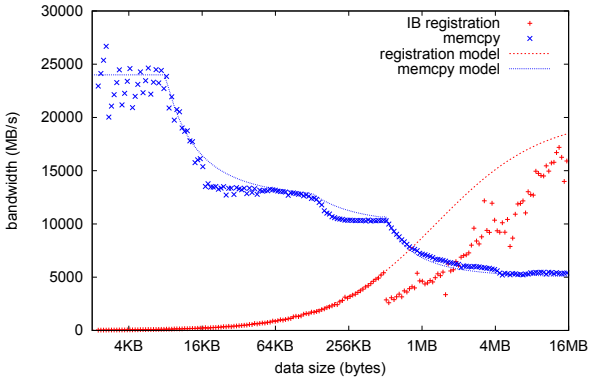

Fig. 1. Registration and memory copy performance comparison on cluster `graphene`.

in InfiniBand must be registered. Two approaches are possible to satisfy this constraint: register memory blocks on the fly; register a buffer at application startup, then copy data into this pre-registered buffer. Memory registration has a significant cost [1] and both approaches have an impact on the overall network performance.

In this paper, we present a software protocol for InfiniBand that copies data through a pre-registered buffer and amortizes the cost of the memory copy by using a superpipeline to overlap copy and RDMA transfer. We propose a performance model of our protocol to study its behavior and optimize its parameters.

The remainder of this paper is organized as follows. In Section 2 we analyze the performance of memory copy and registration. In Section 3 we present an analysis of a pipeline for memory copy. In Section 4, we describe our superpipeline protocol. Section 5 gives benchmarks results. Section 6 compares our work to related works. Section 7 concludes our paper.

2 Performance analysis

In this section, we analyze the performance of memory registration, memory copy, and network transfer, and we propose a performance model.

We run our tests on multiple InfiniBand clusters. Cluster `graphene` features ConnectX DDR (MT26418) cards on quad-core nodes equiped with Intel Xeon X3440. Cluster `infini` has ConnectX2 QDR (MT26428) cards on quad Intel Xeon X5570. To visualize closely what happens, all our graphs use a 5 % increment for message size (i.e. powers of 1.05), not only powers of 2 that hide a lot of details.

Performance of registration. Memory registration in InfiniBand is an expensive operation. We have conducted benchmarks to measure the time consumed

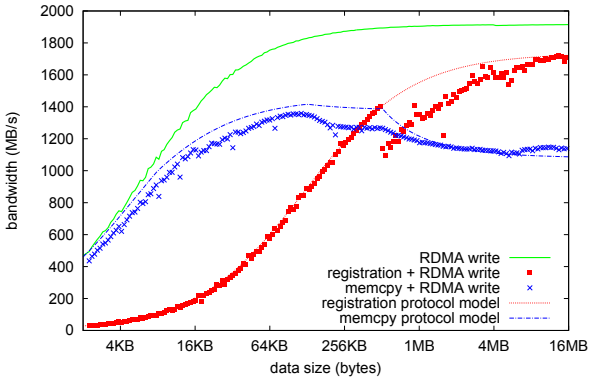

Fig. 2. Impact of registration and memcpy on communication performance on cluster `graphene`.

to register a block of memory on several InfiniBand clusters. For example, the results obtained on cluster `graphene` are depicted in Figure 1, represented as a bandwidth. Let L the length of a given message, we can model the registration time in the form $T_{reg}(L) = \lambda_{reg} + \frac{L}{B_{reg}}$ with λ_{reg} the latency of memory registration, and B_{reg} its bandwidth. On this cluster, we have measured $\lambda_{reg} = 68\,\mu s$. and $B_{reg} = 20\,GB/s$ (actually, $200\,ns$ per 4 KB page). We observe the same order of magnitudes on other DDR and QDR InfiniBand boards.

The cost of registration may have a huge impact on actual communication performance. A naive protocol to send a block of data would consist in dynamically registering the memory region, send the data on the network, then deregister the memory region; the receiver has to perform registration/deregistration too. The performance of such a protocol is depicted in Figure 2 for cluster `graphene`. We observe that the overhead introduced by memory registration lowers the bandwidth by as much as 60 % for packets of roughly 64 KB, and is far from negligible even for larger sizes of messages, with an apparent bandwidth converging asymptotically to $\frac{1}{\frac{1}{B_{net}} + \frac{1}{B_{reg}}}$, which is 91 % of the network bandwidth on our cluster.

Performance of memory copy. Performance of a raw `memcpy` is depicted in Figure 1. The apparent bandwidth decreases when the size of data increases, as a result of cache effects. We can roughly model its behavior with four different bandwidth figures for L1, L2 and L3 caches, and memory. It would require to know actual cache policy and associativity to get a more precise model.

A naive copy-based protocol would copy data on the sender side into a registered memory zone, send the data, then copy the data from the registered memory zone into its final destination in the receiver side. Since memory copies are fast for small messages, this copy-based protocol is usually used for small

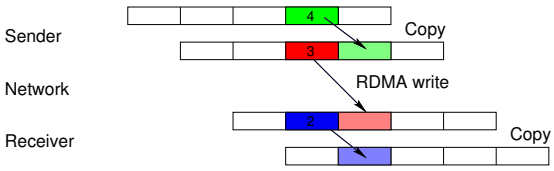

Fig. 3. Pipeline for memory copy: sender copies chunk 3 while sending chunk 4; receiver copies chunk 2 while receiving chunk 3.

messages sent eagerly. Larger messages are usually [2] sent with a *rendezvous* protocol to avoid copies that would lower the available bandwidth.

The apparent bandwidth of the naive copy-based protocol is depicted in Figure 2, converging asymptotically to $\frac{1}{\frac{1}{B_{net}}+\frac{2}{B_{copy}}}$, which is 57 % of the network bandwidth on our cluster.

Real behavior of *rcache*. To amortize the cost of memory registration, it has been proposed a *pin-down cache* [3], or as commonly called today, a *registration cache* (in short: *rcache*). It means that the sender does not unregister the memory zone after a message is sent, in case the same zone is sent again. However it requires a lot of care to be correct [4]. One must use `malloc` hooks, `libc` symbol interception through `LD_PRELOAD` or kernel patches, to invalidate the cache (unregister memory) when memory is deallocated. These mechanisms are not quite portable and may break in subtle ways when interacting with various versions of `libc`, Fortran or OpenMP runtimes, or any runtime that supplies its own memory allocator.

It must be noted that *rcache* does not increase performance by itself. The first send exhibits the same performance as the naive registration-based protocol. Only the subsequent sends of the *same* memory zone will be faster, at the nominal speed of the network. The real world performance of *rcache* depends on the buffer reuse scheme of the application, and may obviously vary from one application to another. For example in NAS Parallel Benchmarks, `SP` and `CG` have 99 % cache hits, `IS` has less than 5 % cache hits, and `LU` sends mostly small messages not concerned by *rcache*. Our goal is to improve performance of `IS` without degrading performance of other benchmarks.

3 Pipelining memory copy

In this section, we study an InfiniBand software protocol which manages memory registration by copying data into a pre-registered buffer instead of dynamically registering data in place.

Since copy and RDMA may be overlapped, a *pipeline* may be used to amortize the cost of the memory copy. Both operations share the same memory bus, but experiments show that copies have a negligible impact on an overlapped RDMA

— memory bandwidth is high these days —, while the copy is slowed down by no more than the bandwidth used by the network. As depicted in Figure 3, each message is divided into *chunks* of a given size. Then on the sender side, we overlap the RDMA transfer of one chunk with the memory copy of the next chunk. Since we use RDMA write, on the receiver side nothing has to be done to make the overlapping happen.

Cost analysis. Let L be the message length, and C the chunk size. For convenience, we assume L is a multiple of C. To model the network with multiple chunks and overlap, we use a model close to LogP [5] with the following notations. Let λ_{net} be the network latency (the L of LogP) and B_{net} the network bandwidth, then we have $T_{net}(L) = \lambda_{net} + L/B_{net}$ as end-to-end transfer time for a raw RDMA write, assuming data is already registered. Let g be the *gap* between messages, then we have $T_{net}(L_1, L_2) = \lambda_{net} + \frac{L_1}{B_{net}} + g + \frac{L_2}{B_{net}}$ as transfer time for two packets of length L_1 and L_2. Let o be the *overhead* for sending messages, namely the CPU time needed to initiate the RDMA operation, that will not be available for overlapping.

In Section 2, we have shown that `memcpy` bandwidth depends on message length. Let $B_{copy}(L)$ be the copy bandwidth for a message of length L. We must notice that since the bandwidth depends on cache effects, the length to take into account is the whole data set, namely L, not C. We assume $\lambda_{copy} = 0$. Then we have $T_{copy}(L) = L/B_{copy}(L)$ as time to copy data of length L.

Then the time for the full pipelined transfer is comprised of: copy of the first chunk: $\frac{C}{B_{copy}(L)}$; steady state with $\frac{L}{C}$ chunks copied and sent: $\frac{L}{C} \times \left(g + \frac{C}{B_{net}}\right)$; the network latency: λ_{net}; copy of the last chunk: $\frac{C}{B_{copy}(L)}$. Therefore the total time of pipelined transfer is:

$$T_{pipeline}(L, C) = \frac{2 \times C}{B_{copy}(L)} + \frac{L}{C} \times g + \lambda_{net} + \frac{L}{B_{net}} \tag{1}$$

Compared to a raw RDMA write, the overhead for the pipeline is $\frac{2 \times C}{B_{copy}(L)} + \frac{L}{C} \times g$. It is comprised of the copy of the first chunk on the sender side, the copy of the last chunk on the receiver side, and the gaps.

Optimal pipeline. We can find the optimal value for C the chunk size. When we draw a graph of $C \to T_{pipeline}(L, C)$ for any fixed L and realistic values of B_{copy} and g, we can see this function admits a minimum. If we assume C to be real instead of integer to make the function differentiable, the derivative with respect to C for a given message length L is:

$$T'_{pipeline}(C) = \frac{2}{B_{copy}(L)} - \frac{L}{C^2} \times g \tag{2}$$

Let C_{opt} be the the optimal chunk size for a given message length L. It corresponds to the zero of the derivative. We solve the equation and get:

$$C_{opt}(L) = \sqrt{\frac{L \times g \times B_{copy}(L)}{2}} \tag{3}$$

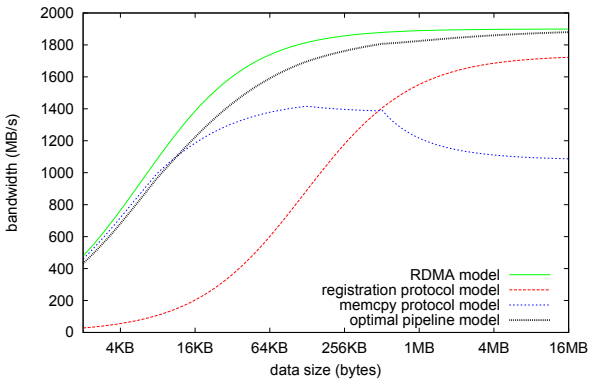

Fig. 4. Bandwidth model for pipeline using parameters from cluster `graphene`.

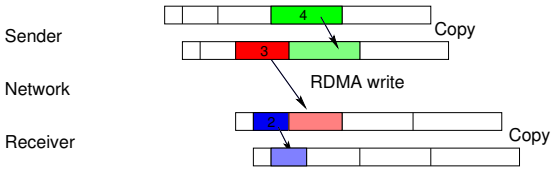

Fig. 5. Super-pipeline for memory copy: a pipeline with a variable chunk size.

Using our performance models for network and copy, we estimate the performance of the pipeline with optimal chunk size, depicted in Figure 4. The performance increase compared to naive protocols is huge, but bandwidth is still lower than raw InfiniBand RDMA and may still be improved. We can see that for messages smaller than 16 KB, the naive copy-based protocol is faster than the optimal pipeline; when computing the optimal chunk size for these messages, we get an optimal with less than one chunk per message, which is wrong. Our hypothesis of C being real instead of integer works only for messages large enough.

4 Optimizations beyond pipeline: superpipeline

In this Section, we present various mechanisms to improve the performance of our protocol, beyond the vanilla pipeline with optimal chunk size presented in the previous Section.

Super-pipeline to lower the number of gaps. Memory copy has a higher bandwidth than RDMA write over the network, as measured in Section 2. Therefore, when pipelining, for each chunk `memcpy` finishes earlier than the RDMA

write for the previous chunk. Thus we propose to increase the chunk size from chunk to chunk while the pipeline is running. We call this mechanism a *superpipeline*, like superpipelines in CPU architecture. This superpipeline mechanism is depicted in Figure 5. It is expected to have a lower number of gaps than the plain pipeline, thus reducing the overhead of the protocol. We must define a suitable progression rule for the chunk size. Let C_i be the chunk size at step i. We may compute the sequence that enables a full overlap of `memcpy` and RDMA; such a sequence will have the fewest gaps. It is defined as:

$$\frac{C_{i+1}}{B_{copy}(L)} = \frac{C_i}{B_{net}} + g - o \tag{4}$$

in other words the time to copy chunk C_{i+1} may be as high as the time to send C_i on the network, including g the gap between packets, but excluding the non-overlapable overhead o, with o and g as defined in Section 3. Since o and g are of the same order of magnitude, $g - o$ is at most in the order of $100\,ns$ and may be neglected compared to the other terms when C_i is several kilobytes. Then Equation 4 simplifies as $C_{i+1} = C_i \times \frac{B_{copy}(L)}{B_{net}}$. Therefore, the general term of the sequence is:

$$C_i = C_0 \times q^i \quad \text{with } q = \frac{B_{copy}(L)}{B_{net}} \tag{5}$$

To compute the protocol overhead, we need to compute the number of gaps. Let n be the number of gaps for a given message of length L. Then L, as the sum of all chunks, is a finite geometric series:

$$L = \sum_{i=0}^{n-1} C_i = C_0 \times \sum_{i=0}^{n-1} q^i = C_0 \times \frac{1-q^n}{1-q} \tag{6}$$

We then solve this equation to get n the number of gaps:

$$n = \log_q \left(1 + \frac{L}{C_0}(q-1)\right) \tag{7}$$

Therefore the total transfer time of our superpipeline protocol is:

$$T_{superpipeline}(L) = \frac{C_0 + C_n}{B_{copy}(L)} + n \times g + \lambda_{net} + \frac{L}{B_{net}} \tag{8}$$

It is very similar to the cost of plain pipeline given in Equation 1, except that the number of gaps is $O(L)$ for fixed-chunk pipeline, $O(\sqrt{L})$ for pipeline with the optimal chunk size C_{opt} as defined by Equation 3, and is lowered to $O(\log(L))$ for superpipeline.

Sub-blocking to lower last chunk copy overhead. The transfer time for the superpipeline given in Equation 8 includes $\frac{C_n}{B_{copy}(L)}$ the time needed to copy the last chunk at the receiver side. Given the general term of the sequence given in Equation 5, C_n is expected to be quite large.

To amortize the cost of the copy at the receiver side, we propose a *sub-blocking* mechanism — namely, a pipeline in the pipeline — to overlap the RDMA and the `memcpy` of the *same chunk*. Among the possible strategies [6] of the receiver side to detect the arrival of RDMA data, we chose to poll a flag at a known memory location. The receiver sets it to zero; the sender writes a 1 through RDMA.

Our *sub-blocking* mechanism consists in dividing each chunk into blocks of a given size b. Every block is comprised of data payload and a flag indicating the presence of data. All the blocks that form a chunk are sent through a single RDMA write. The receiver is then able to detect and consume blocks as they arrive, only one block behind the one being written by the NIC.

With such a method, the cost of the copy at the receiver side is at most the copy of a block $\frac{b}{B_{copy}(L)}$. This methods adds flags in every blocks, which increases the size of packets sent on the network and must be taken into account. However, if we take for example $b = 4\,\text{KB}$ (page size and multiple of MTU) and a flag on a 64-bit word (to avoid atomicity issues depending on endianness), then the overhead is less than $0.2\,\%$.

Overlap *rendezvous* to lower first chunk copy overhead. The overhead of our superpipeline protocol includes the cost of the memory copy for the first chunk C_0. We propose to overlap this copy with the *rendezvous* to lower its impact on performance.

We have shown in Section 3 that the optimal number of chunks is 1 for small messages. The fastest option to send small messages is the naive copy-based protocol. It means that we will use pipeline (or superpipeline) protocols only for large messages, that are sent using a *rendezvous* mechanism to ensure that data is received in place. It must be noted that, although our protocol involves a copy, the *rendezvous* is still relevant because our protocol copies data on the fly and needs to know where to store data.

The cost of a *rendezvous* is twice the latency, namely $2 \times \lambda_{net}$. We propose that the sender copies C_0 the first chunk of the message while the *rendezvous* round-trip takes place. If $\frac{C_0}{B_{copy}} < 2 \times \lambda_{net}$, then the copy of C_0 is free. To maximize overlap in the common case, we chose to use: $C_0 = B_{copy} \times 2\lambda_{net}$. With common contemporary hardware, we get values from $8\,\text{KB}$ to $16\,\text{KB}$ for C_0.

Pipeline folding using N-buffering. In our previous descriptions of our superpipeline, we have assumed that the preregistered buffer is as large as the message. However, registered memory is a finite resource and cannot be arbitrarily large. Therefore we *fold* our superpipeline to fit statically-allocated buffers. Since at any given time, one buffer is copied while another buffer is sent (or received), we can then make our superpipeline a *double-buffering* algorithm. A flow control mechanism is needed to make sender and receiver synchronize their buffer swaps. Since such synchronization through network has a significant latency, we loosen the coupling between the sender and the receiver with triple-buffering. Before

it may send chunk C_i, the sender does not have to wait for the C_{i-1} acknowledgment — that may arrive late because of network latency — but for the C_{i-2} acknowledgement. Moreover, folding the superpipeline in a smaller workspace than the full message length improves cache reuse, although the precise impact is hard to model and to predict.

With all the heuristics and optimizations applied, the total transfer time of our superpipeline protocol is:

$$T_{superpipeline}(L) = \frac{b}{B_{copy}(L)} + g \times \log_q \left(1 + \frac{L}{C_0}(q-1)\right) + \lambda_{net} + \frac{L}{B_{net}} \quad (9)$$

The overhead for to copy of the last sub-block b is low, and the number of gaps is $O(log(L))$.

Discussion. The plain *pipeline* is easy to model and we have determined analytically its optimal chunk size, as we did in Section 3. However, the performance of *superpipeline* depends on a sequence, not a single value, which makes it difficult to solve analytically. Optimization depends on a lot of parameters, that's why we used heuristics (maximize overlap) to determine q and C_0.

Our model is not so precise and some behaviors are hard to predict (cache effects depend on cache policy, associativity, data alignment). The theoretical optimal is thus not necessarily optimal once implemented. However when we trace $C_0 \rightarrow T_{superpipeline}(L, C_0)$ for a given L, variations are *small* around the optimal. We conclude that not-so-precise tuning works well with superpipeline, which is confirmed by experience. Hardwired C_0=12 KB and $q = 1.5$ gives results almost as good as optimal values.

5 Benchmarks

In this Section, we present benchmarks of our superpipeline protocol and compare it against other protocols and implementations.

Raw protocol benchmark. We have implemented our superpipeline protocol in a test program to evaluate its behavior regardless of any other implementation artefact. The results of a ping-pong bandwidth test on clusters `graphene` and `infini` are depicted in Figure 6. We observe the superpipeline is much faster than naive registration and memcpy-based protocols, and is very close to the raw RDMA performance. The overhead compared to raw RDMA is 15 % for 16 KB messages, and less than 5 % for messages larger than 64 KB. However this test program does not implement *rendezvous*, thus cannot overlap C_0 copy and *rendezvous*; real-life overhead is then expected to be lower.

MPI micro-benchmarks. We have then implemented our superpipeline protocol as a driver in our NewMadeleine [7] communication library, which already

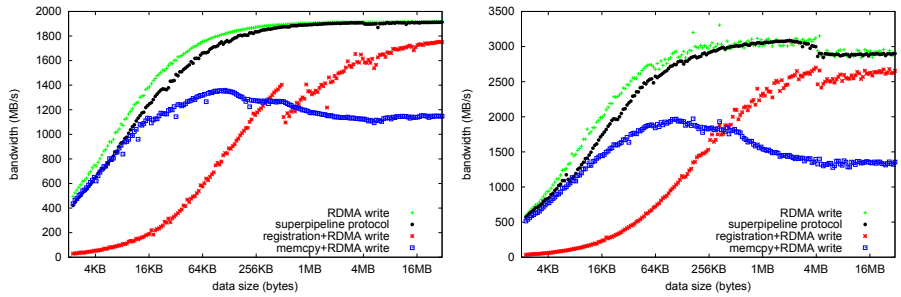

Fig. 6. Raw superpipeline protocol performance on cluster `graphene` (left) and `infini` (right).

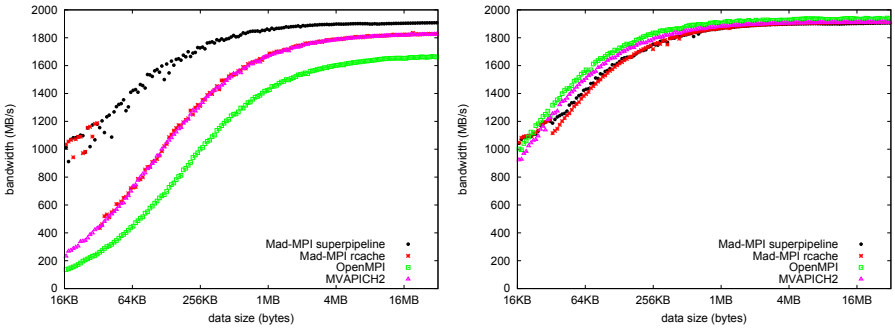

Fig. 7. MPI performance on cluster `graphene` on first touch (left), and best time (right).

has an *rcache* method for InfiniBand, and run MPI benchmarks using its Mad-MPI [8] interface. NewMadeleine has a 32 KB *rendezvous* threshold, and uses plain copy for small messages (no pipeline, no *rcache*). We compare Mad-MPI against OpenMPI-1.4.3 and MVAPICH2-1.6rc2. The results of a MPI ping-pong bandwidth test on cluster `graphene` are depicted in Figure 7. The benchmarks performs 100 round-trips for each message size; we draw separate graphs for the *first* and for the *best* round-trip. We observe the superpipeline gets roughly the same performance as Mad-MPI *rcache best* time, very similar to OpenMPI and MVAPICH2 *best* time. However, when we compare the performance of the *first* round-trip, we observe that Mad-MPI *rcache*, OpenMPI and MVAPICH2 all get low performance because of registration, whereas Mad-MPI superpipeline is unaffected. The superpipeline gets its best performance already at the first send; the others, relying on *rcache*, get poor performance on first send.

MPI NAS Parallel Benchmarks. We have run some benchmarks from the NAS Parallel Benchmarks 3.3.1 on cluster `graphene`. On tests `sp.B.8`, `lu.B.8`

and `cg.C.8`, Mad-Mpi superpipeline and *rcache* get the same performance, 3 % slower than OpenMPI and MVAPICH2, explained by the fact that Mad-Mpi has a slightly higher latency. However, on `is.C.8` Mad-Mpi superpipeline is 9 % faster than Mad-Mpi *rcache* (respectively 3.05 s and 3.32 s), but slightly slower than MVAPICH2 (3.01 s) and OpenMPI (2.92 s).

It means that superpipeline actually improves performance over *rcache* on `IS` which is cache-unfriendly, but some work has to be done in Mad-Mpi to improve latency for small message to get overall better performance.

6 Related works

People working on Mpi implementations and other communication libraries have already studied InfiniBand memory registration and proposed solutions to amortize its cost. Memory registration performance has been analyzed [1] and communication performance modeled [9] without proposing solution to improve performance. Various caching strategies [3, 10, 4] have been proposed, as well as protocols to overlap *rendezvous* and registration [11] in case of cache miss; however, all these solutions exhibits the pitfalls of cache-based strategies.

It has been proposed in OpenMPI [12] to pipeline registration; our model shows that pipelining copies gives better performance than pipelining registration. For the InfiniBand device for MPICH2 [2] and the BCopy mode of SDP [13], it has been investigated to use a copy pipeline with fixed-size chunks. Performance was not convincing, because at that time memory bandwidth was not significantly higher than network bandwidth; our proposal goes further with a superpipeline rather than a flat pipeline, and our theoretical study shows that it works because $B_{copy} > B_{net}$, which has become the common case nowadays with contemporary CPUs and their integrated memory controlers.

7 Conclusion and future works

Memory registration has a major impact on performance for InfiniBand networks. In this paper, we have proposed performance models for InfiniBand network, copies, and registration, and used them to analyze and optimize the performance of a protocol that uses pipelined copy to bring data into registered memory. We have proposed an alternative called *superpipeline* that reduces the number of gaps, and some optimization mechanisms to reduce the cost of the first and last chunk copy. We have implemented and benchmarked our superpipeline protocol, and observed that it gets roughly the same performance as *rcache*-based protocol on cache-friendly communication patterns, and better performance on cache-unfriendly patterns.

As a future work, we will study adaptive strategies to automatically tune the protocol parameters to the machine it is running, and to monitor *rcache* misses/hits to dynamically choose between strategies depending on observed application behavior.

Acknowledgements. This work was supported in part by the ANR-JST project FP3C and the ANR project COOP. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

1. Mietke, F., Rex, R., Baumgartl, R., Mehlan, T., Hoefler, T., Rehm, W.: Analysis of the memory registration process in the mellanox infiniband software stack. In Nagel, W., Walter, W., Lehner, W., eds.: Euro-Par 2006 Parallel Processing. Volume 4128 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2006) 124–133
2. Liu, J., Jiang, W., Wyckoff, P., Panda, D.K., Ashton, D., Gropp, W., Buntinas, D., Toonen, B.: Design and implementation of mpich2 over infiniband with rdma support. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 04). (2004)
3. Tezuka, H., O'Carroll, F., Hori, A., Ishikawa, Y.: Pin-down cache: a virtual memory management technique for zero-copy communication. In: Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP). (April 1998) 308 –314
4. Ohio, P.W., Wyckoff, P., Wu, J.: Memory registration caching correctness. In: IEEE International Symposium on Cluster Computing and the Grid(CCGrid), IEEE Computer Society (2005)
5. Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken, T.: Logp: towards a realistic model of parallel computation. In: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming. PPOPP '93, New York, NY, USA, ACM (1993) 1–12
6. Gupta, R., Tipparaju, V., Nieplocha, J., Panda, D.K.: Efficient barrier using remote memory operations on via-based clusters. In: Cluster'02. (September 2002)
7. Aumage, O., Brunet, E., Furmento, N., Namyst, R.: Newmadeleine: a fast communication scheduling engine for high performance networks. In: CAC 2007: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2007. (March)
8. Trahay, F., Denis, A., Aumage, O., Namyst, R.: Improving reactivity and communication overlap in mpi using a generic i/o manager. In: EuroPVM/MPI, Springer (2007)
9. Hoefler, T., Mehlan, T., Mietke, F., Rehm, W.: Logfp – a model for small messages in infiniband. In: IPDPS. (2006)
10. Bell, C., Bonachea, D.: A new dma registration strategy for pinning-based high performance networks. In: Parallel and Distributed Processing Symposium, 2003. Proceedings. International. (2003) 10 pp.
11. Ou, L., He, X., Han, J.: An efficient design for fast memory registration in rdma. Journal of Network and Computer Applications **32** (2009) 642–641
12. Woodall, T., Shipman, G., Bosilca, G., Graham, R., Maccabe, A.: High performance rdma protocols in hpc. In: Euro PVM/MPI. Volume 4192 of LNCS. (2006)
13. Goldenberg, D., Kagan, M., David, R., Tsirkin, M.S.: Zero copy sockets direct protocol over infiniband – preliminary implementation and performance analysis. In: Symposium on High-Performance Interconnects (HOTI'05). (2005)