Compiler Optimizations for Non-Contiguous Remote Data Movement

Timo Schneider[1], Robert Gerstenberger[2], and Torsten Hoefler[1]

[1] Department of Computer Science, ETH Zurich, Switzerland
{timos,htor}@inf.ethz.ch
[2] Technische Universität Chemnitz, Germany
gerro@hrz.tu-chemnitz.de

Abstract. Remote Memory Access (RMA) programming is one of the core concepts behind modern parallel programming languages such as UPC and Fortran 2008 or high-performance libraries such as MPI-3 One Sided or SHMEM. Many applications have to communicate non-contiguous data due to their data layout in main memory. Previous studies showed that such non-contiguous transfers can reduce communication performance by up to an order of magnitude. In this work, we demonstrate a simple scheme for statically optimizing non-contiguous RMA transfers by combining partial packing, communication overlap, and remote access pipelining. We determine accurate performance models for the various operations to find near-optimal pipeline parameters. The proposed approach is applicable to all RMA languages and does not depend on the availability of special hardware features such as scatter-gather lists or strided copies. We show that our proposed superpipelining leads to significant improvements compared to either full packing or sending each contiguous segment individually. We outline how our approach can be used to optimize non-contiguous data transfers in PGAS programs automatically. We observed a 37% performance gain over the fastest of either packing or individual sending for a realistic application.

1 Introduction

Communication of non-contiguous data is of utmost importance for real application performance. The traditional approach, called "packing" is to copy non-contiguous data into a single contiguous buffer that is then communicated over the network. This practice originated in times where the network was orders of magnitude slower than local processing and copying. However, today, local copies (read and write from/to main memory on one machine) are only slightly faster than remote copies using remote direct memory access (RDMA) over high-performance interconnects (that offer read from main memory at the source machine and write to main memory at the target machine).

The significance of RDMA networking goes beyond the higher bandwidth. It also motivates new Remote Memory Access (RMA) programming models (e.g., UPC [21], Fortran 2008 Coarrays [14], or MPI-3 One Sided [13]) that allow full

exploitation of the new hardware. RMA programming exposes the direct memory access to the user who can issue remote memory writes and reads directly. In addition, such RMA programs are easier to analyze by compilers than message passing programs because the complex message matching problem [4] does not apply (each remote access specifies the target buffer explicitly). This motivates us to explore automatic optimizations, such as pipelining and partial packing, for remote memory accesses.

We now demonstrate a typical parallel application using a simple two-dimensional Laplacian stencil example. The serial version iterates over a two-dimensional array and computes the value of each point from the old value at that point and the old value at the neighboring points (aka. "five-point stencil"). A two-dimensional decomposition for distributed memory parallelism requires communication at the boundaries of each process. Depending on the array layout in memory, one or more directions of communication will access non-contiguous data.

For example, if matrices are stored in row-major order, then data exchanged in the north-south direction is *contiguous* in local memory, while data exchanged in east-west direction is *non-contiguous*. More formally, we can describe any transfer of k Bytes (in total) as a set of k pairs (s_i, d_i) where $1 \leq i \leq k$. Each pair describes a single Byte of the transfer, which is copied from the address s_i at the sender to d_i at the receiver. Without changing the semantics of the transfer, we can sort the pairs, using s_i as a key in ascending order. A transfer is contiguous if $(\forall i \in \{1, \ldots, k\} : s_i = d_i + s_1 - d_1) \wedge (\forall i \in \{1, \ldots, k-1\} : s_i = s_{i+1} - 1)$, otherwise, it is non-contiguous.

Programmers often pack data for all communication directions in order to retain easy maintainability and portability of their code. The following listing shows pseudo-code for the communication part of the Laplacian stencil application:

```
1  for(int iters=0; iters<niters; iters++) {
     compute_2d_stencil(array, ...);
3    // swap arrays (omitted for brevity)
     for(int i=0; i<bsize; ++i) sbufnorth[i] = array[i+1,1];
5    // ... omitted south, east, and west pack loops
     RMA_Put(sbufnorth, rbufnorth, bsize, north);
7    // ... omitted south, east, and west communications
     RMA_Fence();
9    for(int i=0; i<bsize; ++i) array[i+1,0] = rbufnorth[i];
     // ... omitted south, east, and west unpack loops
11 }
```

The loop at line 4 exemplifies the packing of data from the array (potentially non-contiguous) into sbufnorth, a contiguous buffer. The contiguous buffer is then communicated at line 6 (`RMA_Put` represents the language-specific remote write, e.g., assignment to a shared pointer in UPC). The call to `RMA_Fence` represents the language-specific synchronization method (e.g., `upc_fence`).

As mentioned before, similar packing loops can be found in most parallel distributed memory applications, for example WRF [20], MILC [3], NAS LU, MG, SP and BT [22], and SPECFEM3D_GLOBE [6]. In the following we will not differentiate between packing and unpacking as they are symmetric — with "packing" we refer to both packing and unpacking.

If copy overheads (in time and energy) have to be avoided, then one could simply issue all the contiguous pieces using a separate transfer for each. This is exemplified in the following pseudo-code for the same Laplacian application:

```
1 for(int iter=0; iter<niters; ++iter) {
    compute_2d_stencil(array, ...);
3   // swap arrays (omitted for brevity)
    for (int i=0; i<bsize; i++) {
5     RMA_Put(array[i+1, 1], array[i+1, 0], size, north);
      // ... omitted south, east, and west communications
7   }
    RMA_Fence();
9 }
```

Instead of packing the array using a pack loop, all consecutive blocks are sent separately in the loops around lines 4 and following. We call this approach *maximal block* communication. However, sending many small pieces (e.g., a single floating point number in our example) can be very inefficient due to fixed overheads for each transfer.

In this work, we demonstrate how *partial packing* combined with (super)pipelining can improve the communication performance of many scientific codes significantly. Figure 1 provides a high-level overview.

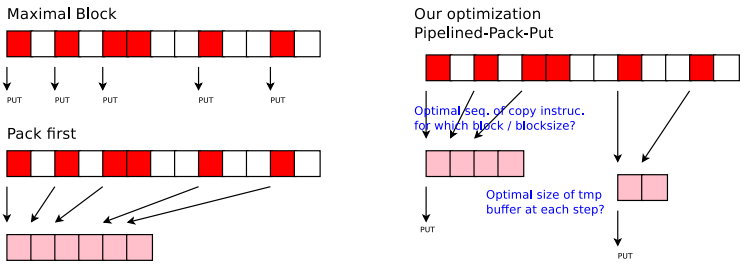

Fig. 1. Methods for sending non-contiguous data in one-sided programming models.

Explicit Datatype Specification Some programming environments offer high-level abstractions for specifying non-contiguous data accesses. MPI, for example, allows the specification of datatypes that simplify and optimize non-contiguous communications. We have shown in a previous study that runtime compilation techniques can speed up the packing of MPI DDTs by a factor of seven [19], and therefore make it competitive with manual packing. The proposed techniques in this work automatically overlap packing and communication to enable further

optimization. In addition, most RMA programming models do not support explicit datatype specifications making our technique necessary for optimizations.

Even if explicit datatype specification is offered, users tend to utilize pack loops [18]. One can go as far and argue that explicit data-access declarations are not necessary because copy loops and other communication constructs can be easily identified using static analysis and transformed into more efficient representations. For example, Kjolstad et al. demonstrated a simple static analysis that detects and optimizes common pack loops [12].

Our work applies to both, library implementations and compiled code. However, we argue that (super)pipelining techniques are most efficient when the communication and partial packing can be integrated into the application computation. In this work, we step into this direction by modeling the optimization of non-contiguous transfers by pipelining and overlapping packing and sending.

The detailed contributions of this paper are the following:

- We show how a compiler can generate an instruction sequence for near-optimal copying of data into a temporary buffer. Our tuned copy code is up to two times faster than copy functions such as bcopy and memcpy. The resulting code shall be inlined as partial pack-code.
- We show how modeling communication and copy performance can be utilized to transform a sequence of communication and pack statements into an efficient pipelined schedule for a near-optimal combination of packing and communicating.

2 Pipelining for non-contiguous Put operations

In the rest of this paper we assume that we have a set S whose elements are tuples of the form (s, r, l). Each element of this set describes one block of data which is l Bytes in size and resides on the sender at s and has to be transferred to the receiver at address r. Furthermore we assume that S is *minimal*, that means there exists no set S' that describes the same data-movement pattern with a smaller number of elements in the set.

A minimal set S can be constructed by simulating a program execution. Each put operation would be recorded as one tuple of the set K. The set K can be minimized to S using the following procedure: The tuples in K are sorted according to their s elements and elements are checked pair-wise in the sorted list. If $(s_i + l_i = s_{i+1}) \wedge (r_i + l_i = r_{i+1})$ then we can combine the tuples i and $i+1$ into a new tuple $(s_i, r_i, l_i + l_{i+1})$. This procedure is repeated until a fixpoint is reached. This can be extended to symbolic analysis, for example, by using abstract interpretation [7].

Maximal block communication would now put every block (as identified by a tuple) separately. Let the cost to issue a single put operation of length l be $T_{put}(l)$, and let $x.l$ identify the l element in tuple x. The overall cost of maximal block communication is:

$$T = \sum_{a \in S} T_{put}(a.l)$$

Another option would be to search for certain features in the set S and exploit them. One such feature is that, while the data is non-contiguous at the sender, it is actually placed in a consecutive buffer at the receiver. A very common example for this is the transposition of a matrix which is distributed across multiple processes. Such a communication pattern is required in multi-dimensional FFT codes and seismic wave propagation codes, such as SPECFEM3D_GLOBE [18]. In such a case, instead of sending each element of S individually, we could also copy all elements into a single temporary buffer on the sender side, and transfer this buffer to the receiver using a single RMA put operation. In that case the cost for the entire transfer would not only depend on the performance of the put, but also of the copy operation, which we denote as $T_{copy}(l)$. The overall cost of this scheme can therefore be expressed as:

$$T = \sum_{a \in S} T_{copy}(a.l) + T_{put}\left(\sum_{b \in S} b.l\right)$$

When compared to the first scheme, it is clear that the second one can only be faster if the difference between many small put operations and one big put operation is big enough to offset the time required for the copy operations. This has been exploited before [9], especially by systems which perform message vectorization. Small transfers attain a smaller bandwidth than bigger ones, due to a constant latency and send overhead. In Figure 2 we plot the time it takes to send 800 KB of data, with a different number of put operations, so that the size of each put varies between 8 and 1000 Bytes.

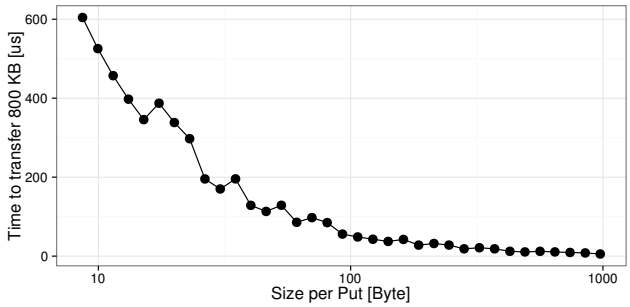

Fig. 2. Varying the number (and therefore the size) of put operations to transfer 800 KB of data shows that bigger puts are more efficient than small ones.

We can see that if the transfer is realized with small puts, i.e., one double precision floating point value per put, it takes 110 times longer to transfer the data then using puts of size 1 KB. The reason is that each put operation has some constant overhead on the host CPU and also on the Network Interface Card (NIC). It seems like minimizing the number of put operations can lead to higher performance. However, the benchmark above only considered communication (on the NIC and CPU) and no parallel packing (on the CPU).

The LogGP network model [1] models CPU and NIC overheads separately as o and g. Those two terms may overlap and a put operation can be performed in $\max(o, g)$. Therefore to minimize the total transfer time T it is beneficial to overlap some of the packing with put operations. This can be done by partitioning the data movement operation expressed by the set S into a series of (non-empty) partitions P_i ($1 \leq i \leq n$, we assume that elements in S can be split into multiple pieces that belong to different partitions P_i). Copying the first partition P_1 into the temporary buffer cannot be overlapped, similarly the sending of the last partition P_n cannot be overlapped. Therefore the total time can be expressed as:

$$T = \sum_{a \in P_1} T_{copy}(a.l) + \sum_{i=2}^{n} \max\left(T_{put}\left(\sum_{b \in P_{i-1}} b.l \right), \sum_{c \in P_i} T_{copy}(c.l) \right) + T_{put}\left(\sum_{d \in P_n} d.l \right)$$

Fixed pipeline To minimize the total transfer time we would need to minimize this function over all possible partitionings of S. This optimization problem can be solved with traditional optimization methods or heuristics. A simple heuristic would be to fix the size of the put operations we want to perform, and partition S in such a way that all puts (except the last) are of this size. We call this method the *fixed pipeline* method.

Superpipelining The simple fixed-size-put scheme can be improved by increasing the size of each partition P_i as we progress. The rationale for this is that we should keep the size of the first put operation low, as the copy operations before it cannot be overlapped with anything. On the other hand we want to minimize the total number of puts. If we assume that $\forall s : T_{copy}(s) < T_{put}(s)$ we can increase the size of each put. The goal is to keep the network (which is then the bottleneck of the transfer) saturated. This will be the case if we ensure that the time to pack the data for the i-th put operation is smaller than the time taken to perform the $i-1$st put operation with which the copying is overlapped. To compute the optimal pipeline, we need to know the functions $T_{put}(s)$ and the inverse of the function $T_{copy}(s)$, since we want to know how many Bytes we can copy for the next put. This approach of gradually increasing the size of pipeline stages to achieve optimal overlap and throughput is called superpipelining.

In the following we show a semi-analytic performance model for the performance of RDMA put operations to get an approximation for $T_{put}(s)$. Unfortunately the performance of copy operations cannot be modeled that nicely due to the vast number of influencing factors (cache state, cache sizes, cache associativity, instruction choice for the copy operation, unrolling of copy operations, etc.). Therefore we propose a method to generate a fast copy code, which at the same time gathers performance measurements which can be used to approximate the inverse of $T_{copy}(s)$.

3 Data Movement Operations

Modern CISC architectures offer a plethora of instructions capable of copying data in main memory. For this study we focus on x86-64, since it is the most prevalent architecture in parallel computing today. On modern x86-64 architectures copying data between memory locations can be done in two different ways. Most data-movement instructions only copy between registers and memory, therefore a copy between memory locations consists of two parts: Copying the data from memory into a register and copying it back from the register into a different memory location. In addition, the movs instruction family is able to copy directly from memory to memory. There are many different ways how to copy data in and out of a register. Perhaps the most well known one is the mov instruction family (this includes all variants of the mov instruction for different widths, i.e., movb to copies a single Byte, movw copies two Bytes, movl copies four Bytes and movq copies eight Bytes). Being a CISC design, the x86 instruction set also includes specialized instructions to copy strings: the load-string lods and store-string instruction family stos. They essentially behave like the mov instruction, however, the programmer is free to choose where he places operands for the mov instructions, those use the registers %rsi, %rdi for the source and destination address and use %rax as temporary buffer. All those instruction can only operate on up to eight Byte at a time. With the SIMD extensions (i.e., SSE2 and AVX) load/store instructions became available that are able to load/store 16 (SSE2) or 32 (AVX) Byte from/to a register in one instruction.

SIMD instructions offer another interesting set of features to the programmer: Not only can loads and stores be performed using much wider registers, but also special loads and stores are offered for aligned data. Another novelty is the introduction of non-temporal stores, which bypass the cache and write directly into memory. Of course, writing directly into memory is much slower than writing into the cache. However, when copying large blocks of data (larger than the last level cache) it is useless to write any (but the last chunk) of data to the cache, since this data will be evicted from the cache anyway by later writes. Knowingly bringing useless data into the cache is of course suboptimal, since it inflicts additional overhead because of the cache coherency protocol. Therefore temporal store instructions also have to be considered carefully.

Another important choice the programmer (or compiler) has to make when writing a copy-loop is the choice of the loop instructions he uses. When data is copied using a movs instruction, a loop can be formed by simply prefixing this instruction with the rep prefix. This prefix repeats the prefixed instruction until the %rcx register is zero and decrements the %rcx register after each iteration. The direction of operation (decrementing or incrementing %rsi and %rdi) is set with the direction flag. Of course the rep prefix is only an option when the movs instruction is used, as all other alternatives require more than one instruction to perform a memory to memory copy operation. For those cases we again have multiple options: We can use the loop instruction, which jumps to a label if %rcx is not zero and decrements this register before each jump. However, with this variant we have to adjust the value of the source and destination pointers

manually in the loop. The third option is to manually do a comparison at the end of the loop body and then use an instruction of the jmp family to jump to the start of the loop, depending on the result of the comparison.

Figure 3 gives an overview over the possibilities of combinations of data-movement and loop forming instructions offered by the x86-64 instruction set. Another variable the programmer has to consider is the unrolling factor of the copy loop: the overhead of the branching instruction can be alleviated by performing several copy operations inside of the loop body. However, since copy operations are memory bandwidth bound, too much unrolling can also be detrimental to the performance since loading of the instruction stream also creates memory pressure.

The x86-64 instruction set offers even more data-movement instructions (i.e., push/pop, compare-and-swap) which are not considered here since they are specialized instructions with more functionality than data movement and should therefore always be slower than the simpler instructions.

We optimize the code used for copying automatically, using algorithm outlined in Figure 4 select the optimal combination of data-movement instruction and unroll factor combination for selected block sizes.

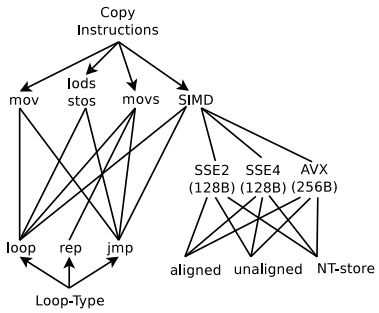

Fig. 3. Data-movement and loop-forming instructions on x86-64.

for all $s \in \{2^0, 2^1, ..., 2^{20}\}$ **do**
  for all $i \in \{\text{copy inst.} \times \text{loop type}\}$ **do**
    for all $f \in \{2^0, 2^1, ..., 2^{10}\}$ **do**
      $t_i^f \leftarrow$ time to copy s Byte with instruction i, unrolled f times (median of 1000 runs)
    end for
  end for
  copyroutine$(s) \leftarrow \min(t_i^f)$
end for

Fig. 4. Algorithm used to generate optimized copy code.

For each block size all possible combinations of instruction(-swidth) and unroll factor is computed, since not every combination supports all sizes. The measurement of the performance of each combination is repeated several times (1,000 times in our case) and for each combination the median of those times is computed. The optimal combination of instruction and unroll factor for a given block size is then chosen.

This algorithm is performed for a number of sizes and assuming the source data is in cache or not in cache. We tuned only sizes up to one Megabyte because our superpipelining does not require larger blocks. The gathered information is then used to construct a near-optimal sequence of CPU instructions to perform the copy for packing.

The performance of our copy code, which we call *fcopy* is shown in Figure 5. We compare it to the memcpy() and bcopy() function. We optimize for two cases: (1) the source data resides in cache ("Cache Hot") and (2) the source data needs

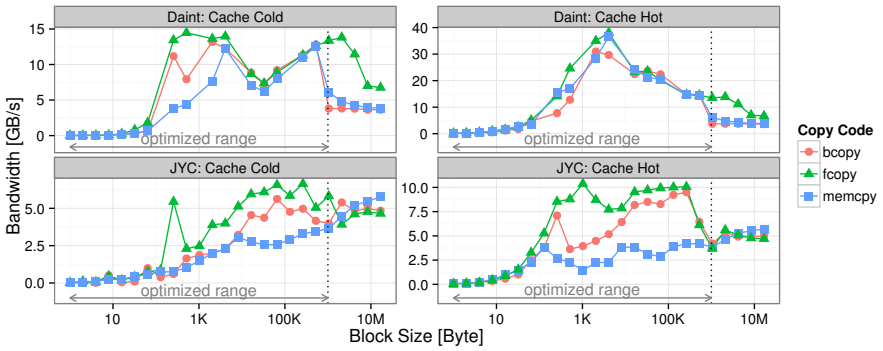

Fig. 5. Performance of our copy code *fcopy* compared to the performance of memcpy and bcopy on JYC and Daint. Our optimized code is up to seven times faster than memcpy and up to 2.6 times faster than bcopy. Note that we optimized the code for block sizes up to 1 MB (left of the dotted line).

to be loaded from main memory ("Cache Cold"). We assume that a compiler analysis could determine the reuse distance of the to-be-copied data and decide on the best instruction sequence.

For the performance data presented in this paper, we use two different machines: JYC, the Blue Waters test system at the National Center for Supercomputing Applications, which consists of a single cabinet Cray XE6 (approx. 50 nodes with 1,600 Interlagos 2.3 — 2.6 GHz cores) and Piz Daint, a Cray XC30 at CSCS with dual-socket 8-core 64-bit Intel SandyBridge CPUs clocked at 2.6 GHz.

4 Modeling Communication

To be able to model the performance of one-sided non-contiguous data transfers, we need to model not only the performance of the local copying of data, but also the performance of the remote memory copies.

One-sided data transfers follow the same general scheme, independent of the actual API in which they are implemented: A synchronization epoch is started, then a number of remote memory operations is started, after which the synchronization epoch is again closed. Those operations are combined into a single statement if synchronization is not relaxed. In our case however, the goal is to overlap packing with RMA operations. Thus, we utilize a relaxed synchronization model for our communication. In this model the time to execute n put operations, with sizes s_i can be modeled as $t = L + n \times o_{put} + G \sum_{i=1}^{n} s_i$. This model is similar to the LogGP model [1]. The constant overhead (latency, synchronization overhead) is denoted as L, where o_{put} is the overheads for the put operation, which is independent of the size of the data buffer being transferred. In LogGP we would differentiate between the overhead on the NIC, g and the overhead

on the host CPU o, however, in practice these values are hard to measure independently. Therefore we model $\max(o, g)$ as o_{put}. The inverse bandwidth of the transfer is expressed by G.

We parametrize this model by performing between 1 and 50 puts in a loop, each with the same data buffer size. The data buffer size is varied between a single Byte and 800 KB. Then we fit the above model to the measured data. Each measurement is repeated 50 times, and we use the median value to minimize the effects of outliers due to noise.

The results of these measurements on JYC are plotted in Figure 6 for out-of-cache inter-node communication. We focus on inter-node communication in this work, because on our test system intra-node communication is handled by copying the data directly from the target to the destination buffer, using the XPMEM [23] kernel module to access another processes address space. Since this copy operation is not performed in an extra progression thread, overlapping intra-node communication is therefore not possible.

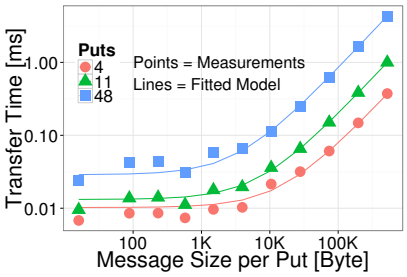

Fig. 6. Inter-node communication performance on JYC. If we parametrize the model with this data we get: $L = 1\,\mu s$, $o = 0.686\,\mu s$, $G = 0.17\,ns/B$.

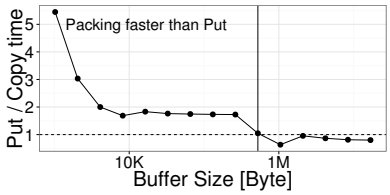

Fig. 7. Relationship between put and copy performance for different block sizes. Below 10 KB puts are much more expensive than additional copies, while above 500 KB it is slower to pack more data than to send it over the network directly.

If we use the data collected on JYC to parametrize our model we get $L = 1\,\mu s$, $o = 0.68\,\mu s$ ($0.44\,\mu s$ for in-cache data), $G = 0.17\,ns/B$. This model fits the measured data quite well, the R^2 value is 0.999. This means 99.9% of the variance observed in the data is explained by the model. For Daint the values are $L = 1\,\mu s$, $o = 0.66\,\mu s$ (same for in-cache data), $G = 0.6\,ns/B$ and an R^2 of 0.979.

We can now use the performance model for communication and the data collected during the construction of the copy method to determine optimal sizes for the partitions of S which are transferred with a single put operation, and by which factor we can increase this size for consecutive puts. To do this we look at the quotient $r = \frac{T_{put}(s)}{T_{copy}(s)}$ for different sizes s. This ratio is plotted in Figure 7.

If $r < 1$ it means that we should never copy this much data into a temporary buffer, the put will be faster than copying this data was, therefore we will not enlarge the partitions of S beyond this point. If we plot that ratio for our test system, we can see that for 500 KB and larger, collecting more data (for a larger

put operation) takes longer than sending it immediately, therefore we stop to increase the size of the partitions, once we copied a block of size 500 KB. Note that $T_{copy}(s)$ is an upper bound for the performance of filling the temporary buffer for a put of size s. The real performance of this operation is $\sum_{a \in P_i} T_{copy}(a.l)$ which can be much lower in case of very small consecutive blocks in the data layout on the sender side, therefore the start value and the rate at which to increase partition sizes have to be computed for each data layout. Furthermore we can see that below 10 KB puts are much more expensive than additional copies (the ratio is above 2), therefore it would be inefficient to perform smaller puts. Because of that we start our superpipeline protocol with an initial partition size of at least 10 KB.

5 Results

In this section we will demonstrate the performance of our optimization with two examples. The first example is the matrix transpose part of an FFT code. The consecutive blocks on the sender side are (depending on the total size) between 128 and 1792 Byte in size. On the receiver the data is stored in one contiguous buffer. The stride between the blocks on the sender side (for a given total size) is constant.

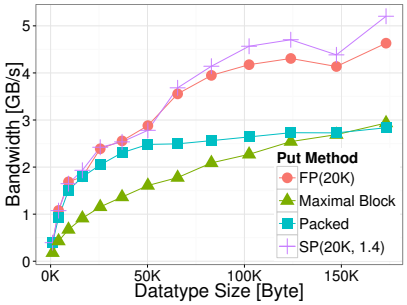

Fig. 8. Performance of different pipelining approaches on JYC for an FFT code. Consecutive blocks at the sender grow with the problem size from 128 Byte blocks to 1792 Byte.

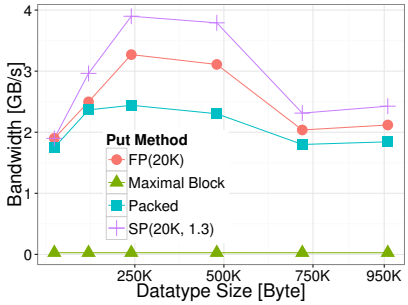

Fig. 9. Performance of different pipelining approaches on JYC for a data transfer in SPECFEM3D_GLOBE. For this transfer, each block is 12 Byte in size.

We show the performance of the three strategies to transfer non-contiguous data with RMA put operations explained in this paper in Figure 8. When each consecutive block is transferred individually, the achieved bandwidth is very low, and grows for bigger problem instances due to the growth of the size of the consecutive blocks. This method is labeled as *Maximal Block* (cf. Figure 1). The performance of this approach can be improved considerably by packing data on the sender, prior to sending it. We can either pack all data and send it with one put operation (labeled as *Packed*), or overlap packing and put operations. If all (except the last) chunks have the same size, 20 KB in our example, labeled

as *FP(20K)* for fixed-pipeline we can improve the total bandwidth by 1.3 GB/s (42%, compared to Maximal Block). Our superpipeline protocol can achieve an additional performance increase for the larger problem instances of about 652 MB/s (13%, compared to FP) when the size of each put is increased by a factor of 1.3, while the first put is again 20 KB in size. This variant is labeled as *SP(20K, 1.3)* where SP stands for superpipeline protocol.

In Figure 9 we conduct the same experiment with a send data layout from the SPECFEM3D_GLOBE seismic wave propagation simulation code [6]. Here the individual consecutive blocks are much smaller, only 12 Bytes in size, and their size remains constant when the problem size is increased. The sender stride between blocks is 24 Byte while the data is put into a contiguous buffer at the receiver. Since the blocks are so small, *Maximal Block* performs much worse than in the FFT example. In this plot, one can clearly see that the performance gain attainable by superpipelining depends heavily on the speed of the copy operations. As long as the extent of the data layout fits (together with the pack buffer) in the 2 MB L2 cache, the performance of superpipelining is much better than that of fixed size pipelining. After that point their performance becomes similar again. Superpipelined packing is 148 times faster than sending each block individually, up to 37% faster than packing everything into one block before sending and 17% faster than fixed-pipeline packing. To show the portability of our method we conduct the same experiments on Daint, a Cray XC30 with Intel SandyBridge CPUs. The results are plotted in Figures 11 and 12. In Figure 10 we perform the same comparison with another data layout present in the SPECFEM3D_GLOBE code, where the data blocks on the sender are not stored with a regular stride, but in a irregular fashion (indexed type in MPI). Each block is four Byte in size and the data is stored consecutively on the receiver. Because of the small block size and the resulting high copy overhead, the difference between the packing methods is small. Superpipelining is 118 times faster than *Maximal Block*, 18% faster than *Packed* and up to 8% faster than fixed size pipelining.

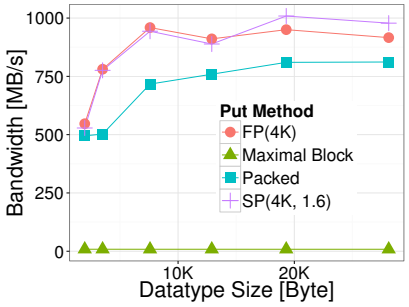

Fig. 10. Performance comparison of an irregular data transfer on JYC in SPECFEM3D_GLOBE. The blocks at the sender are 4 Byte in size.

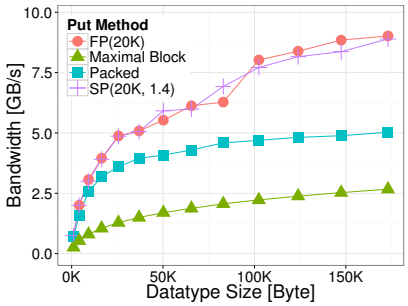

Fig. 11. Performance of different pipelining approaches on Daint for an FFT code. Consecutive blocks at the sender grow with the problem size from 128 Byte blocks to 1792 Byte.

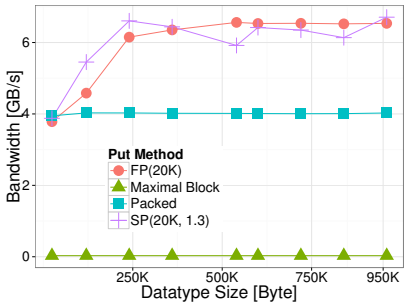

Fig. 12. Performance of different pipelining approaches on Daint for a data transfer in SPECFEM3D_GLOBE. For this transfer, each block is 12 Byte in size.

6 Related Work

Many compiler optimizations are based on peephole optimization techniques: Matching rules are applied to an intermediate compilation result and if a match is found, the code is replaced with a functionally equivalent, but faster, alternative. Those transformations are often created manually by domain experts. However, approaches where the optimization opportunities are automatically generated, similar to our approach of optimizing copy code, also have been suggested. Superoptimization [2] tries to optimize an instruction sequence by generating *all* possible instruction sequences up to a certain length and checking if they are functionally equivalent to the target instruction sequence. The problem with superoptimization is the exponential growth of the search space with the length of the considered instruction sequence and the number of available instructions. The interest in superoptimization also seems to have become smaller since this technique has been initially proposed — to the best of our knowledge, there is no publicly available superoptimizer which includes the whole instruction set (incl. AVX, SSE4, etc.) of a modern x86 CPU. Recent applications of superoptimization techniques [17], use heuristics to keep the search space manageable.

Superpipelining [8] was first proposed to overlap memory registration with RDMA operations. We extended this technique for copying non-contiguous data. Santhanaraman et al. [16] suggested to use gather/scatter support offered by modern network stacks [5,15] to implement MPI datatypes for two-sided point to point transfers and collectives. Since MPI allows that sender and receiver specify different datatypes in the respective send or receive call, this information (the datatype layout) has to be communicated first. After that the non-contiguous data is sent using an InfiniBand gather operation [15]. At the receiver, the data is stored (possibly in a different layout) using an InfiniBand scatter operation. In this work we are focusing on the one-sided programming model, where the sender has complete knowledge over the data layout at the receiver. Furthermore we do not rely on special hardware features for the transfer. The problem of transferring

data does not only occur in message passing and RDMA programming, but also when programming for accelerators, which have a private memory, such as GPUs. Jablin et al. [10] for example strive to optimize CPU to GPU communication by using compiler passes and a run time layer which optimize the scheduling of the communication, i.e., achieve communication-computation overlap by early binding. Jenkins et al. [11] propose GPU kernels to pack MPI datatypes, which gives large improvements over packing them with the host CPU.

Schneider et al. [19] also optimized the packing of MPI derived datatypes. MPI DDTs are traditionally interpreted at runtime, which is often slower than manual pack loops written for a specific case and optimized by the compiler at compile time. We mitigate that by generating machine code to pack MPI DDTs at runtime. This increased packing performance by up to a factor of seven. However, none of the pipelining techniques described in this work have been used, the generated pack function pack the complete message into a buffer before sending.

7 Conclusions

In this work we showed which optimizations a compiler for partitioned global address space languages can perform, in order to accelerate non-contiguous data transfers, without the requirement of special purpose hardware. We showed two main targets for optimization: the scheduling of RMA put operations and the instruction sequence used to copy small chunks of data into a temporary buffer for sending them. We show an algorithm to optimize the copy code and show that the resulting code outperforms readily available compiler builtins such as memcpy and system functions such as bcopy. We show how pipelining copying data and transferring it can improve performance and how we can leverage performance models of the network operations, as well as performance data of the copy code to choose suitable parameters for the suggested pipelining protocols. All optimizations can be implemented in compilers for PGAS languages or RMA libraries using well-known techniques.

Acknowledgments

We thanks the Swiss National Supercomputing Center (CSCS) and the Blue Waters project at NCSA/UIUC for access to the test systems. We also thank the anonymous reviewers for comments that greatly improved our work.

References

1. Alexandrov, A., Ionescu, M.F., Schauser, K.E., Scheiman, C.: LogGP: Incorporating long messages into the logP model – one step closer towards a realistic model for parallel computation. In: Proceedings of the 7th annual ACM symposium on Parallel algorithms and architectures (SPAA'95). pp. 95–105 (1995)
2. Bansal, S., Aiken, A.: Automatic generation of peephole superoptimizers. ACM SIGPLAN Notices 41(11), 394–403 (2006)

3. Bernard, C., Ogilvie, M., et al.: Studying quarks and gluons on MIMD parallel computers. Int. J. of High Performance Computing Applications 5(4), 61–70 (1991)
4. Bronevetsky, G.: Communication-sensitive static dataflow for parallel message passing applications. In: Proceedings of the 7th annual IEEE/ACM international symposium on Code generation and optimization (CGO'09) (2009)
5. ten Bruggencate, M., Roweth, D.: DMAPP - an API for one-sided program models on Baker systems. In: Cray user group conference, (CUG'10) (2010)
6. Carrington, L., Komatitsch, D., et al.: High-frequency simulations of global seismic wave propagation using SPECFEM3D_GLOBE on 62K processors. In: Proceedings of the 22nd international conference on Supercomputing (SC'08) (2008)
7. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'77). pp. 238–252 (1977)
8. Denis, A.: A high performance superpipeline protocol for InfiniBand. In: Proc. of the European conference on Parallel Processing. pp. 276–287 (2011)
9. Hiranandani, S., Kennedy, K., Tseng, C.W.: Evaluating compiler optimizations for Fortran D. J. of Parallel and Distributed Computing 21(1), 27–45 (1994)
10. Jablin, T.B., Prabhu, P., Jablin, J.A., Johnson, N.P., Beard, S.R., August, D.I.: Automatic CPU-GPU communication management and optimization. ACM SIGPLAN Notices 46(6), 142–151 (2011)
11. Jenkins, J., Dinan, J., et al.: Enabling fast, noncontiguous GPU data movement in hybrid MPI + GPU environments. In: Proceedings of the IEEE international conference on Cluster computing (CLUSTER'12) (2012)
12. Kjolstad, F., Hoefler, T., Snir, M.: Automatic datatype generation and optimization. In: Proceedings of the 17th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'12). pp. 327–328 (2012)
13. MPI Forum: MPI: A Message-Passing Interface Standard. Version 3
14. Numrich, R.W., Reid, J.: Co-array Fortran for parallel programming. SIGPLAN Fortran Forum 17(2), 1–31 (1998)
15. Pfister, G.F.: High Performance Mass Storage and Parallel I/O, chap. An introduction to the InfiniBand architecture, pp. 617–632 (2001)
16. Santhanaraman, G., Wu, J., Panda, D.K.: Zero-copy MPI derived datatype communication over InfiniBand. In: Recent advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI'04). pp. 47–56 (2004)
17. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: Proceedings of the international conference on Architectural support for programming languages and operating systems (ASPLOS'13). pp. 305–316 (2013)
18. Schneider, T., Gerstenberger, R., Hoefler, T.: Application-oriented ping-pong benchmarking: How to assess the real communication overheads. J. of Computing (May 2013)
19. Schneider, T., Kjolstad, F., Hoefler, T.: MPI datatype processing using runtime compilation. In: Proc. of EuroMPI'13 (Sep 2013)
20. Skamarock, W.C., Klemp, J.B.: A time-split nonhydrostatic atmospheric model for weather research and forecasting applications. J. Comput. Phys. 227(7), 3465–3485 (3 2008)
21. UPC Consortium: UPC language specifications. Version 1.2 (2005)
22. der Wijngaart, R.F.V., Wong, P.: NAS parallel benchmarks version 2.4. Tech. rep., NAS Technical Report NAS-02-007 (2002)
23. Woodacre, M., Robb, D., Roe, D., Feind, K.: The SGI AltixTM 3000 global shared memory architecture (2005)