Computing
DOI 10.1007/s00607-013-0330-4

Application-oriented ping-pong benchmarking: how to assess the real communication overheads

Timo Schneider · Robert Gerstenberger · Torsten Hoefler

Received: 16 December 2012 / Accepted: 27 April 2013

Abstract Moving data between processes has often been discussed as one of the major bottlenecks in parallel computing—there is a large body of research, striving to improve communication latency and bandwidth on different networks, measured with ping-pong benchmarks of different message sizes. In practice, the data to be communicated generally originates from application data structures and needs to be serialized before communicating it over serial network channels. This serialization is often done by explicitly copying the data to communication buffers. The message passing interface (MPI) standard defines derived datatypes to allow zero-copy formulations of non-contiguous data access patterns. However, many applications still choose to implement manual pack/unpack loops, partly because they are more efficient than some MPI implementations. MPI implementers on the other hand do not have good benchmarks that represent important application access patterns. We demonstrate that the data serialization can consume up to 80 % of the total communication overhead for important applications. This indicates that most of the current research on optimizing serial network transfer times may be targeted at the smaller fraction of the communication overhead. To support the scientific community, we extracted the send/recv-buffer access patterns of a representative set of scientific applications to build a benchmark that includes serialization **and** communication of application data and thus reflects **all** communication overheads. This can be used like traditional ping-pong benchmarks to determine the **holistic** communication latency and bandwidth

T. Schneider (✉) · T. Hoefler
ETH Zurich, Department of Computer Science, Universitätstr. 6, Zurich 8092, Switzerland
e-mail: timo.schneider@inf.ethz.ch

T. Hoefler
e-mail: torsten.hoefler@inf.ethz.ch

R. Gerstenberger
University of Illinois at Urbana-Champaign, Urbana, IL, USA
e-mail: gerro@illinois.edu

Published online: 09 May 2013

as observed by an application. It supports serialization loops in C and Fortran as well as MPI datatypes for representative application access patterns. Our benchmark, consisting of seven micro-applications, unveils significant performance discrepancies between the MPI datatype implementations of state of the art MPI implementations. Our micro-applications aim to provide a standard benchmark for MPI datatype implementations to guide optimizations similarly to the established benchmarks SPEC CPU and Livermore Loops.

Keywords MPI datatypes · Benchmark · Data movement · Access-pattern

Mathematics Subject Classification 68M10 · 68M14

1 Motivation and state of the art

One of the most common benchmarks in HPC to gauge network performance is a ping-pong benchmark over a range of different message sizes which are sent from and received into a consecutive buffer. With such a benchmark we can judge the minimum achievable latency and maximum available bandwidth for an application. As we show in Fig. 1a, this correlates weakly with the communication overhead that typical computational science applications experience, because such applications generally do not communicate consecutive data, but serialize (often called *pack*) their data into a consecutive buffer before sending it.

The MPI standard [15] is the de facto standard for implementing high-performance scientific applications. The advantage of MPI is that it enables a user to write performance-portable codes. This is achieved by abstraction: Instead of expressing a communication step as a set of point-to-point communications in a low-level communication API it can be expressed in an abstract and platform independent way. MPI implementers can tune the implementation of these abstract communication patterns for specific machines.

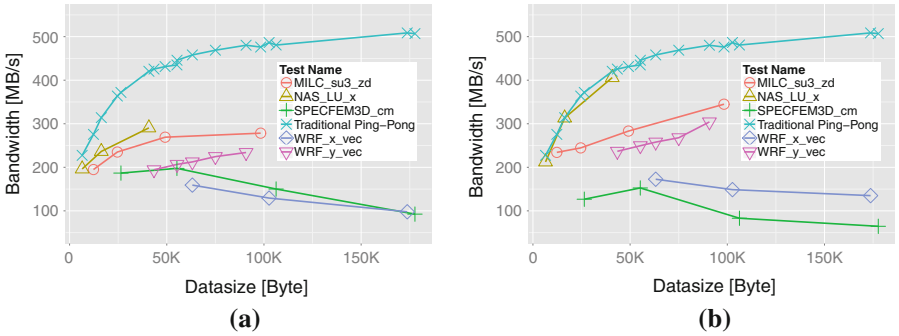

Fig. 1 Bandwidth attained by several applications benchmarks, compared with a normal ping-pong of the same size. No application is able to attain the performance outlined by the standard ping-pong benchmark when manual pack loops are used. MPI datatypes (DDTs) recognize that the buffer in the NAS_LU_x (cf. Sect. 2 for a detailed description of all patterns.) case is already contiguous and do not perform an extra copy. However, there are also many cases where MPI DDTs perform worse than manual packing. **a** Manual packing with Fortran 90. **b** Packing with MPI DDTs

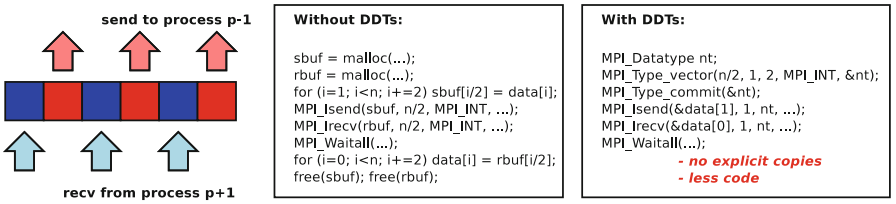

Fig. 2 An example use case for MPI derived datatypes

MPI derived datatypes allow the specification of arbitrary data layouts in all places where MPI functions accept a datatype argument (e.g., MPI_INT). We give an example for the usage of MPI DDTs to send/receive a vector of integers in Fig. 2. All elements with even indices are to replaced by the received data, elements with odd indices are to be sent. Without the usage of MPI DDTs one would either have to allocate temporary buffers and manually pack/unpack the data or send a large number of messages. The usage of MPI DDTs greatly simplifies this example. If the used interconnect supports non-contiguous transfers (such as Cray Gemini [2]) the two copies can be avoided completely. Therefore the usage of MPI DDTs not only simplifies the code but also can improve the performance due to the zero-copy formulation. In Fig. 1b we show that some applications can benefit from using MPI DDTs instead of manual pack loops (for example NAS LU and MILC, as was already demonstrated in [13]).

Not many scientific codes leverage MPI DDTs, even though their usage would be appropriate in many cases. One of the reasons might be that current MPI implementations in some cases still fail to match the performance of manual packing, despite the work that is done on improving DDT implementations [8,20,22]. Most of this work is guided by a small number of micro-benchmarks. This makes it hard to gauge the impact of a certain optimization on real scientific codes.

Coming back to the high-level language analogy made before and comparing this situation to the one of people developing new compiler optimizations techniques or microarchitecture extensions we see that, unlike for other fields, there is no application-derived set of benchmarks to evaluate MPI datatype implementations. Benchmark suites such as SPEC [10] or the Livermore Loops [14] are used (e.g., [1]) to evaluate compilers and microarchitectures. To address this issue, we developed a set of micro-applications[1] that represent access patterns of representative scientific applications as optimized pack loops as well as MPI datatypes. Micro-applications are, similarly to mini-applications [5,7,12], kernels that represent real production level codes. However, unlike mini-applications that represent whole kernels, micro-applications focus on one particular aspect (or "slice") of the application, for example the I/O, the communication pattern, the computational loop structure, or, as in our case, the communication data access pattern.

1.1 Related work

Previous work in the area of MPI DDTs focuses on improving its performance, either by improving the way DDTs are represented in MPI or by using more cache efficient

[1] Which can be downloaded from http://unixer.de/research/datatypes/ddtbench.

strategies for packing and unpacking the datatype to and from a contiguous buffer [8]. Interconnect features such as RDMA Scatter/Gather operations [22] have also been considered. Several application studies demonstrate that MPI datatypes can outperform explicit packing in real-world application kernels such as FFTs [13] and matrix redistribution [4]. However, performance of current datatype implementations remains suboptimal and has not received as much attention as latency and bandwidth, probably due to the lack of a reasonable and simple benchmark. For example Gropp et al. [11] found that several basic performance expectations are violated by MPI implementations in use today, i.e., sending data with a more expressive MPI datatype is in some cases faster than using a less expressive one. The performance of MPI datatypes is often measured using artificial micro-benchmarks, which are not related to specific application codes, such as the benchmarks proposed by Reussner et al. [17]. We identify an unstructured access class, which is present in many molecular dynamics and finite element codes. This access pattern is completely ignored in many datatype optimization papers. However, the issue of preparing the communication buffer has received very little attention compared to tuning the communication itself. In this work, we show that the serialization parts of the communication can take a share of up to 80 % of the total communication overheads because they happen at the sender *and* at the receiver.

In contrast to the related work discussed in this section, our micro-applications offer three important features: (1) they represent a comprehensive set of application use cases, (2) they are easy to compile and use on different architectures, and (3) they isolate the data access and communication performance parts and thus enable the direct comparison of different systems. They can be used as benchmarks for tuning MPI implementations as well as for hardware/software co-design of future (e.g., exascale) network hardware that supports scatter/gather access. This paper is an extended and improved version of [18]. In this version we extended the description of the analyzed application codes and investigated if the extracted access patterns are persistent across the application run. We added results for the datatype performance of Cray's vendor MPI and compare the performance of manual pack loops implemented in Fortran and C.

2 Representative communication data access patterns

We analyzed many parallel applications, mini-apps and application benchmarks for their local access patterns to send and receive memory. Our analysis covers the domains of atmospheric sciences, quantum chromodynamics, molecular dynamics, material science, geophysical science, and fluid dynamics. We created seven micro-apps to span these application areas. Table 1 provides an overview of investigated application classes, their test cases, and a short description of the respective data access patterns. In detail, we analyzed the applications WRF [19], SPECFEM3D_GLOBE [9], MILC [6] and LAMMPS [16], representing respectively the fields of weather simulation, seismic wave propagation, quantum chromodynamics and molecular dynamics. We also included existing parallel computing benchmarks and mini-apps, such as the NAS Parallel Benchmarks (NPB) [21], the Sequoia benchmarks as well as the Mantevo mini-apps [12].

Table 1 Overview of the application areas, test names, and access patterns of the micro-applications contained in our benchmark

Application class	Testname	Access pattern
Atmospheric science	WRF_x_vec	Struct of 2d/3d/4d face exchanges in different directions (x,y), using different (semantically equivalent) datatypes: nested vectors (_vec) and subarrays (_sa)
	WRF_y_vec	
	WRF_x_sa	
	WRF_y_sa	
Quantum chromodynamics	MILC_su3_zd	4d face exchange, z direction, nested vectors
Fluid dynamics	NAS_MG_x	3d face exchange in each direction (x,y,z) with vectors (y,z) and nested vectors (x)
	NAS_MG_y	
	NAS_MG_z	
	NAS_LU_x	2d face exchange in x direction (contiguous) and y direction (vector)
	NAS_LU_y	
Matrix transpose	FFT	2d FFT, different vector types on send/recv side
	SPECFEM3D_mt	3d matrix transpose, vector
Molecular dynamics	LAMMPS_full	Unstructured exchange of different particle types (full/atomic), indexed datatypes
	LAMMPS_atomic	
Geophysical science	SPECFEM3D_oc	Unstructured exchange of acceleration data for different earth layers, indexed datatypes
	SPECFEM3D_cm	

Those applications spend a significant amount of their run-time in communication functions, for example MILC up to 12 %, SPECFEM3D_GLOBE up to 3 %, and WRF up to 16 % for the problems we use in our micro-applications, which is confirmed by the analysis done in [3] and [9].

We found that MPI DDTs are rarely used in the HPC codes considered, and thus we analyzed the data access patterns of the (pack and unpack) loops that are used to (de-)serialize data for sending and receiving. Interestingly, the data access patterns of all those applications can be categorized into three classes: *Cartesian Face Exchange*, *Unstructured Access* and *Interleaved Data*.

In the following we will describe each of the three classes in detail and give specific examples of codes that fit each category.

2.1 Face exchange for n-dimensional Cartesian grids

Many applications store their working set in n-dimensional arrays that are distributed across one or more dimensions. In a communication face, neighboring processes then exchange the "sides" of "faces" of their part of the working set. Such access patterns can be observed in many of the NAS codes, such as LU and MG, as well as in WRF and

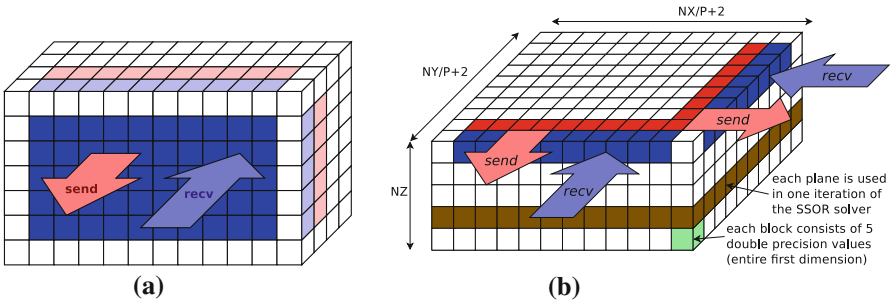

Fig. 3 Data layout of the NAS LU and MG benchmark. **a** NAS MG, **b** NAS LU

MILC. For this class of codes, it is possible to construct matching MPI DDTs using the subarray datatype or nested vectors. Some codes in this class, such as WRF, exchange faces of more than one array in each communication step. This can be done with MPI DDTs using a struct datatype to combine the sub-datatypes that each represents a single array.

The **Weather Research and Forecasting (WRF)** application uses a regular three-dimensional Cartesian grid to represent the atmosphere. Topographical land information and observational data are used to define initial conditions of forecasting simulations. The model solution is computed using a Runge–Kutta time-split integration scheme in the two horizontal dimensions with an implicit solver in the vertical dimension. WRF employs data decompositions in the two horizontal dimensions only. WRF does not store all information in a single data structure, therefore the halo exchange is performed for a number of similar arrays. The slices of these arrays that have to be communicated are packed into a single buffer. We create a struct of hvectors of vector datatypes or a struct of subarrays datatypes for the WRF tests, which are named WRF_{x,y}_{vec,sa}, one test for each direction, and each datatype choice (nested vectors or subarrays, respectively). WRF contains 150 different static datatypes which can be reused during the application run.

NAS MG communicates the faces of a 3d array in a 3d stencil where each process has six neighbors. The data access pattern for one direction is visualized in Fig. 3a. The data-access pattern in MG can be expressed by an MPI subarray datatype or using nested vectors. Our NAS_MG micro-app has one test for the exchange in each of the three directions NAS_MG_{x,y,z} using nested vector datatypes. NAS MG uses a few different but static access patterns and therefore all datatypes can be reused.

The **NAS LU** application benchmark solves a three-dimensional system of equations resulting from an unfactored implicit finite-difference discretization of the Navier–Stokes equations. In the dominant communication function, LU exchanges faces of a four-dimensional array. The first dimension of this array is of fixed size (5). The second (nx) and third (ny) dimension depend on the problem size and are distributed among a quadratic processor grid. The fourth (nz) dimension is equal to the third dimension of the problem size. Figure 3b visualizes the data layout. Our NAS_LU micro-app represents the communication in each of the two directions NAS_LU_{x,y}. NAS LU uses a few different but static access patterns and therefore all datatypes can be reused.

The **MIMD Lattice Computation (MILC)** Collaboration studies Quantum Chromodynamics (QCD), the theory of strong interaction, a fundamental force describing the interactions of quarks and gluons. The MILC code is publicly available for the study of lattice QCD. The su3_rmd application from that code suite is part of SPEC CPU2006 and SPEC MPI. Here we focus on the CG solver in su3_rmd. Lattice QCD represents space-time as a four-dimensional regular grid of points. The code is parallelized using domain decomposition and communicates with neighboring processes that contain off-node neighbors of the points in its local domain. MILC uses 96 different MPI DDTs to accomplish its halo exchange in the 4 directions (named $\pm x, \pm y, \pm z, \pm t$). The datatypes stay the same over the course of the application run. The MILC_su3_zd micro-app performs the communication done for the $-z$ direction.

An important observation we made from constructing datatypes for the applications in the face exchange class is that the performance of the resulting datatype heavily depends on the data layout of the underlying array. For example, if the exchanged face is contiguous in memory (e.g., for some directions in WRF and MG), using datatypes can essentially eliminate the packing overhead completely. That is the reason we included tests for all different directions of each application.

2.2 Exchange of unstructured elements

The codes in this class maintain scatter–gather lists which hold the indices of elements to be communicated. Molecular Dynamics applications (e.g., LAMMPS) simulate the interaction of particles. Particles are often distributed based on their spatial location and particles close to boundaries need to be communicated to neighboring processes. Since particles move over the course of the simulation each process keeps a vector of indices of local particles that need to be communicated in the next communication step. This access pattern can be captured by an indexed datatype. A similar access pattern occurs in Finite Element Method (FEM) codes (e.g., Mantevo MiniFE/HPCCG) and the Seismic Element Method (SEM) codes such as SPECFEM3D_GLOBE. Here each process keeps a mapping of mesh points in the local mesh defining an element and the global mesh. Before the simulation can advance in time the contributions from all elements which share a common global grid point need to be taken into account.

LAMMPS is a molecular dynamics simulation framework which is capable of simulating many different kinds of particles (i.e., atoms, molecules, polymers, etc.) and the forces between them. Similar to other molecular dynamics codes it uses a spatial decomposition approach for parallelization. Particles are moving during the simulation and may have to be communicated if they cross a process boundary. The properties of local particles are stored in vectors and the indices of the particles that have to be exchanged are not known a priori. Thus, we use an indexed datatype to represent this access. We created two tests, LAMMPS_{full,atomic}, that differ in the number of properties associated with each particle. The LAMMPS code in its current form does not amend to datatype reuse.

SPECFEM3D_GLOBE is a spectral-element application that allows the simulation of global seismic wave propagation through high resolution earth models. It is

used on some of the biggest HPC systems available [9]. The earth is described by a mesh of hexahedral volume elements. Grid points that lie on the sides, edges or corners of an element are shared between neighboring elements. SPECFEM3D_GLOBE maintains a mapping between grid points in the local mesh to grid points in the global mesh. Before the system can be marched forward in time, the contributions from all grid points that share a common global grid point need to be considered. The contribution for each global grid point needs to be collected, potentially from neighboring processes. Our micro-app representing SPECFEM3D_GLOBE has two tests, SPECFEM3D_{oc,cm}, which differ in the amount of data communicated per index. The nine different datatypes needed by this code can be reused, since their usage only depends on the used mesh, which does not change during runtime.

The Mantevo mini-app **MiniFE** has a data access pattern very similar to SPECFEM3D_GLOBE, which is not surprising, since MiniFE models a finite element code and the seismic element method in SPECFEM3D_GLOBE is a variant of the FEM method. The Mantevo mini-app MiniMD is a miniature version of the LAMMPS code described above.

Our results show that current MPI DDT implementations are often unable to improve such unstructured access over packing loops. Furthermore, the overhead of creating datatypes for this kind of access (indexed datatypes) is high.

2.3 Interleaved data or transpose

Fast Fourier Transforms (FFTs) are used in many scientific applications and are among the most important algorithms in use today. FFTs can be multi-dimensional: As the one-dimensional Fourier Transform expresses the input as a superposition of sinusoids, the multi-dimensional variant expresses the input as a superposition of plane waves, or multi-dimensional sinusoids. For example, a two-dimensional FFT can be computed by performing 1d-FFTs along both dimensions. If the input matrix is distributed among MPI processes along the first dimension, each process can compute the first 1d-FFT without communication. After this step the matrix has to be redistributed, such that each process now holds complete vectors of the other dimension, which effectively transposes the distributed matrix. After the second 1d-FFT has been computed locally the matrix is transposed again to regain the original data layout. In MPI the matrix transpose is naturally done with an MPI_Alltoall operation.

Hoefler and Gottlieb presented a zero-copy implementation of a 2d-FFT using MPI DDTs to eliminate the pack and unpack loops in [13] and demonstrated performance improvements up to a factor of 1.5 over manual packing. The FFT micro-app captures the communication behavior of a two-dimensional FFT.

SPECFEM3D_GLOBE exhibits a similar pattern, which is used to transpose a distributed 3D array. We used Fortran's COMPLEX datatype as the base datatype for the FFT case in our benchmark (in C two DOUBLEs) and a single precision floating point value for the SPECFEM3D_mt case. The MPI DDTs used in those cases are vectors of the base datatypes where the stride is the matrix size in one dimension. To interleave the data this type is resized to the size of one base datatype. An example for this technique is given in Fig. 4.

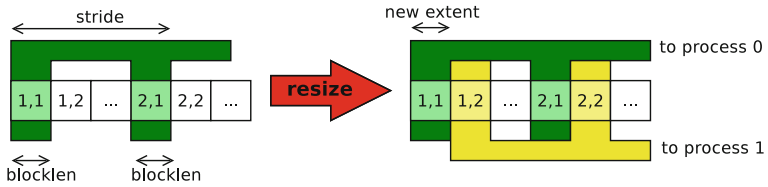

Fig. 4 Datatype for 2d-FFT

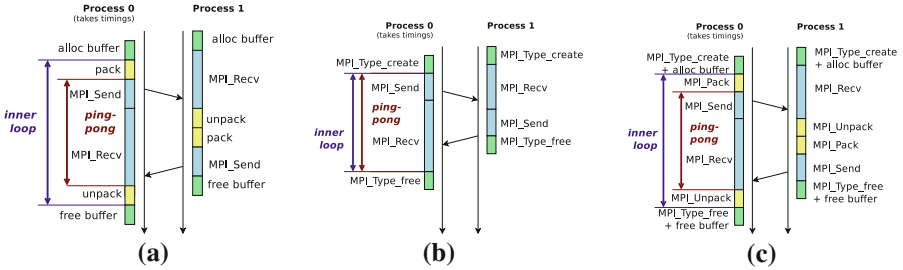

Fig. 5 Measurement loops for the micro-applications. The time for each phase (*rectangle*) is measured on process 0. **a** Manual Pack Loop, **b** Send/Recv with MPI DDTs, **c** MPI_Pack

3 Micro-applications for Benchmarking MPI datatypes

We implemented all data access schemes that we discussed above as micro-applications with various data sizes. For this, we used the original data layout and pack loops whenever possible to retain the access pattern of the applications. We also choose array sizes that are representing real input cases. The micro-applications are implemented in Fortran (the language of most presented applications) as well as C to enable a comparison between compilers. We compiled all benchmarks with highest optimization.

All benchmark results shown in this paper have been obtained on either the **Odin** cluster at IU Bloomington or on **JYC**, the Blue Waters test system at the National Center for Supercomputing Applications. Odin consists of 128 nodes with AMD Opteron 270 HE dual core CPUs and an SDR Infiniband interconnect. JYC consists of a single cabinet Cray XE6 (approx. 50 nodes with 1,600 Interlagos 2.3–2.6 GHz cores). We used the GNU compiler version 4.6.2 and compiled all benchmarks with $-$O3 optimization.

We performed a ping-pong benchmark between two hosts using MPI_Send() and MPI_Recv() utilizing the original pack loop and our datatype as shown in Fig. 5. Our benchmark also performs packing with MPI using MPI_Pack() and MPI_Unpack() (cf. Fig. 5c), however, packing overhead for explicit packing with MPI has been omitted due to lack of space and the small practical relevance of those functions. For comparison we also performed a traditional ping-pong of the same data size as the MPI DDTs type size.

The procedure runs two nested loops: the outer loop creates a new datatype in each iteration and measures the overhead incurred by type creation and commit; the inner loop uses the committed datatype a configurable number of times. In all experiments

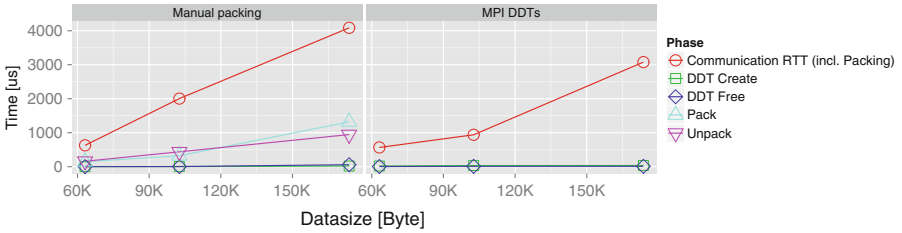

Fig. 6 Median duration of the different benchmark phases for the WRF_x_vec test, using Open MPI 1.6 on Odin

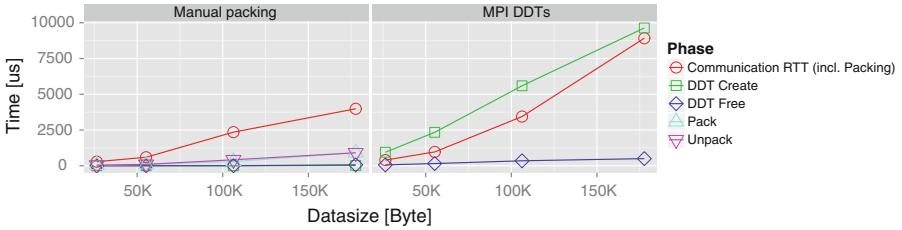

Fig. 7 Median duration of the different benchmark phases for the SPECFEM3D_cm test, using Open MPI 1.6 on Odin

we used 10 outer loop iterations and 20 iterations of the inner loop. Time for each phase (rectangles in Fig. 5) is recorded in a result file. We provide an example script for GNU R to perform the packing overhead analysis as shown in this paper. Measurements are done only on the client side, so the benchmark does not depend on synchronized clocks.

If we measure the time for each phase multiple times (in the two loops described above) and plot the median value for each phase, we get a result as shown in Fig. 6, where we plot the times for three different sizes of the WRF_x_vec test, using Open MPI 1.6. Note that the time for packing has been added to the communication round trip time (RTT) of manual packing to enable direct comparison with the MPI DDT case where packing happens implicitly and is thus also included in the communication time.

It can be seen that using MPI datatypes is beneficial in this case. The WRF application exhibits a face exchange access pattern, as explained before. The datatypes needed for this pattern are simple to create, therefore also the datatype creation overhead is low. For the SPECFEM3D_cm test (Fig. 7) the situation is different: datatypes for unstructured exchanges are very costly to construct. Also none of the MPI implementations was able to outperform manual packing for this test. Unless otherwise noted we assume datatype reuse in all benchmarks. That means the costs for creating and destroying datatypes (or allocating/freeing buffers) are not included in the communication costs. This is reasonable because most applications create their datatypes only once during their entire run and amortize these costs over many communication steps.

For two-dimensional FFTs (Fig. 8), the usage of derived datatypes also improves performance, compared with manual packing. Note the large difference in the times required for manual packing compared to manual unpacking—this is caused by the fact that during packing large blocks can be copied, while during unpack each element has to be handled individually.

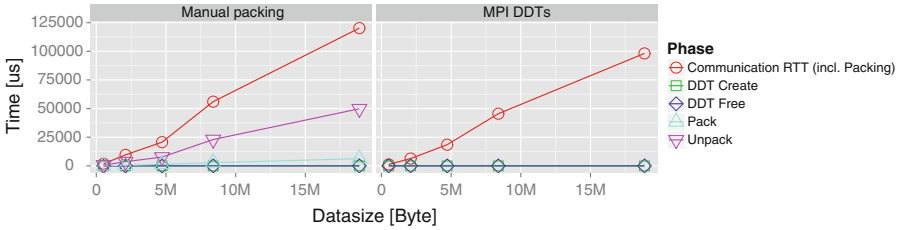

Fig. 8 Median duration of the different benchmark phases for the FFT test, using Cray MPI on JYC (Blue Waters test system)

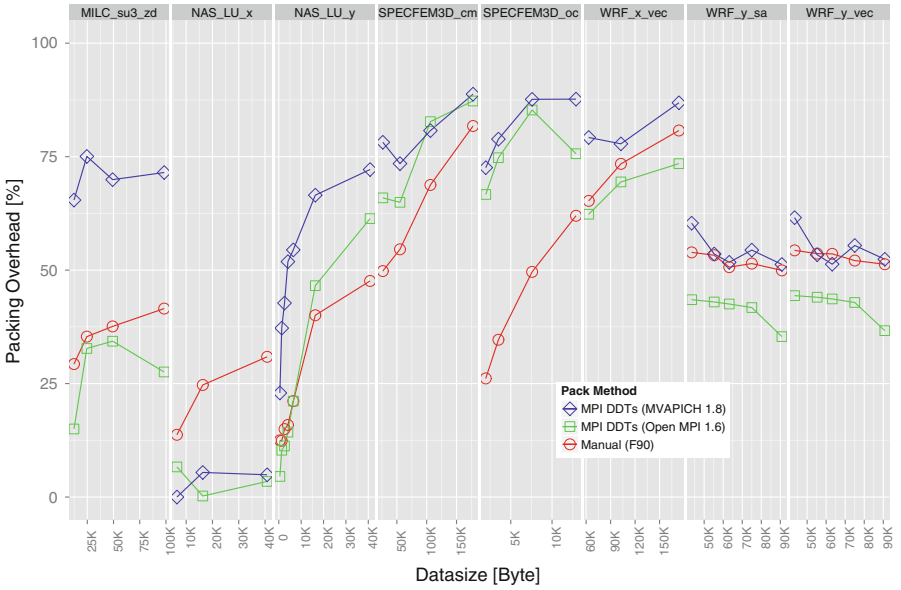

Fig. 9 Packing overheads (relative to communication time) for different micro-apps and datasizes and MPI implementations on Odin

It is interesting to know the fraction of time that data packing needs, compared with the rest of the round-trip. Since we can not measure the time for data-packing directly in the MPI DDT benchmark we use the following method: Let t_{pp} be the time for a round-trip including all packing operations (implicit or explicit) and t_{net} the time to perform a ping-pong of the same size without packing.

The overhead for packing relative to the communication time can be expressed as $ovh = \frac{t_{pp} - t_{net}}{t_{pp}}$.

The serial communication time t_{net} was practically identical for the tested MPI implementations (<5 % variation). This enables us to plot the relative overheads for different libraries into a single diagram for a direct comparison. Figure 9 shows those relative pack overheads for some representative micro-application tests performed with Open MPI 1.6 as well as MVAPICH 1.8 on the Odin cluster; we always ran one process per node to isolate the off-node communication.

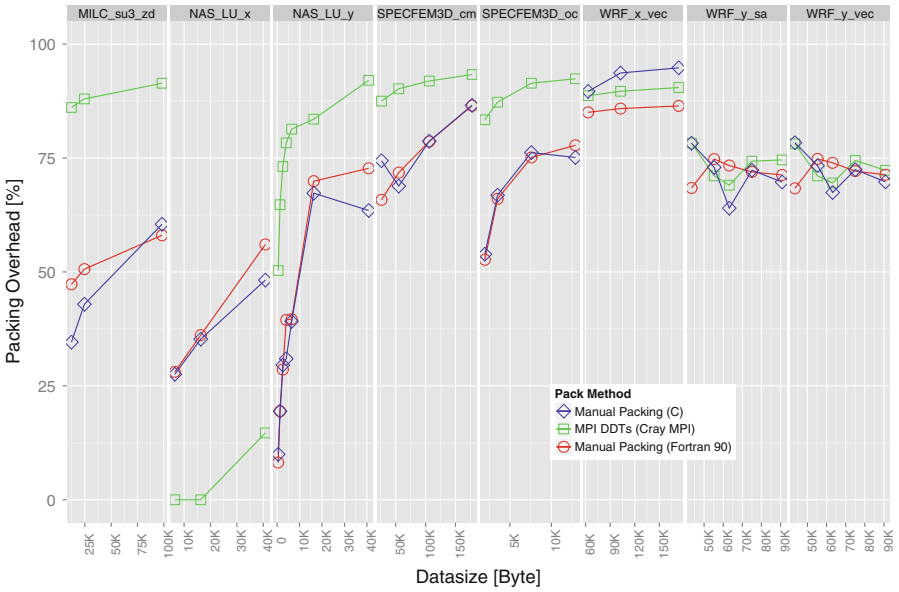

Fig. 10 Packing overheads (relative to communication time) for different micro-apps and datasizes and compilers on JYC

Note that the overhead for the creation of the datatype was not included in the calculations of the packing overheads in Fig. 9, because most applications are able to cache and reuse datatypes. From this figure we can make some interesting observations: In the NAS_LU_x test case both MPI implementations outperform manual packing by far, the packing overhead with MPI DDTs is almost zero. In this case the data is already contiguous in memory, and therefore does not need to be copied in the first place—the manual packing is done anyway in the NAS benchmark to simplify the code. Both MPI implementations seem able to detect that the extra copy is unnecessary. We observe that the datatype engine of Open MPI performs better than MVAPICH's implementation. The SPECFEM3D tests show that unordered accesses with indexed datatypes are not implemented efficiently by both Open MPI and MVAPICH. This benchmark shows the importance of optimizing communication memory accesses: up to 81 % of the communication time of the WRF_x_vec test case are spent with packing/unpacking data, which can be reduced to 73 % with MPI DDTs. In the NAS_LU_x case, which sends a contiguous buffer, using MPI DDTs reduce the packing overhead from 30 to 7 % without increasing the code complexity.

In Fig. 10 we compare the packing overhead of several micro-applications when different compilers are used. We implemented each test in C as well as in Fortran (while most of the original code was written in Fortran). For most tests there is no significant difference. For WRF the packing loop expressed in Fortran is slightly faster. For MILC the packing loop written in C is much faster on JYC. Cray's MPI implementation is outperformed by manual packing in all of our tests. This indicates some optimization potential in the datatype implementation of Cray MPI.

4 Conclusions

We analyzed a set of scientific applications for their communication buffer access patterns and isolated those patterns in micro-applications to experiment with MPI datatypes. In this study, we found three major classes of data access patterns: Face exchanges in n-dimensional Cartesian grids, irregular access of datastructures of varying complexity based on neighbor-lists in FEM, SEM and molecular dynamics codes as well as access of interleaved data in order to redistribute data elements in the case of matrix transpositions. In some cases (such as WRF) several similar accesses to datastructures can be fused into a single communication operation through the usage of a struct datatype. We provide the micro-applications to guide MPI implementers in optimizing datatype implementations and to aid hardware-software co-design decisions for future interconnection networks.

We demonstrated that the optimization of data packing (implicit or explicit) is crucial, as packing can make up up to 80 % of the communication time with the data access patterns of real world applications. We showed that in some cases zero-copy formulations can help to mitigate this problem. Those findings make clear that system designers should not rely solely on ping-pong benchmarks with contiguous buffers, they should take the communication buffer access patterns of real applications into account.

While we present a large set of results for relevant systems, it is necessary to repeat the experiments in different environments. Thus, we provide the full benchmark source-code and data analysis tools at http://unixer.de/research/datatypes/ddtbench/.

Acknowledgments This work was supported by the DOE Office of Science, Advanced Scientific Computing Research, under award number DE-FC02-10ER26011, program manager Sonia Sachs.

References

1. Aiken A, Nicolau A (1988) Optimal loop parallelization. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI'88), vol 23. ACM, pp 308–317
2. Alverson R, Roweth D, Kaplan L (2010) The Gemini System interconnect. In: Proceedings of the IEEE symposium on high performance interconnects (HOTI'10), IEEE Computer Society, pp 83–87
3. Armstrong B, Bae H, Eigenmann R, Saied F, Sayeed M, Zheng Y (2006) HPC benchmarking and performance evaluation with realistic applications. In: SPEC benchmarking workshop
4. Bajrović E, Träff JL (2011) Using MPI derived datatypes in numerical libraries. In: Recent advances in the message passing interface (EuroMPI'11). Springer, Berlin, pp 29–38
5. Barrett RF, Heroux MA, Lin PT, Vaughan CT, Williams AB (2011) Poster: mini-applications: Vehicles for co-design. In: Proceedings of the companion on high performance computing, networking, storage and analysis (SC'11 companion), ACM, pp 1–2
6. Bernard C, Ogilvie MC, DeGrand TA, Detar CE, Gottlieb SA, Krasnitz A, Sugar RL, Toussaint D (1991) Studying quarks and gluons on MIMD parallel computers. Int J Supercomput Appl SAGE 5:61–70
7. Brunner TA (2012) Mulard: a multigroup thermal radiation diffusion mini-application. Technical report, DOE exascale research conference
8. Byna S, Gropp W, Sun XH, Thakur R (2003) Improving the performance of MPI derived datatypes by optimizing memory-access cost. In: Proceedings of the IEEE international conference on cluster computing (CLUSTER'03). IEEE Computer Society, pp 412–419

9. Carrington L, Komatitsch D, Laurenzano M, Tikir M, Michéa D, Le Goff N, Snavely A, Tromp J (2008) High-frequency simulations of global seismic wave propagation using SPECFEM3D_GLOBE on 62k processors. In: Proceedings of the ACM/IEEE conference on supercomputing (SC'08), IEEE Computer Society, pp 60:1–60:11
10. Dixit KM (1991) The SPEC benchmarks. In: Parallel computing, vol 17. Elsevier Science Publishers B.V., Amsterdam, pp 1195–1209
11. Gropp W, Hoefler T, Thakur R, Träff JL (2011) Performance expectations and guidelines for MPI derived datatypes. In: Recent advances in the message passing interface (EuroMPI'11), LNCS, vol 6960. Springer, New York, pp 150–159
12. Heroux MA, Doerfler DW, Crozier PS, Willenbring JM, Edwards HC, Williams A, Rajan M, Keiter ER, Thornquist HK, Numrich RW (2009) Improving performance via mini-applications. Technical report, Sandia National Laboratories, SAND2009-5574
13. Hoefler T, Gottlieb S (2010) Parallel zero-copy algorithms for fast Fourier transform and conjugate gradient using MPI datatypes. In: Recent advances in the message passing interface (EuroMPI'10), LNCS, vol 6305. Springer, New York, pp 132–141
14. McMahon FH (1986) The livermore Fortran kernels: a computer test of the numerical performance range. Technical report, Lawrence Livermore National Laboratory, UCRL-53745
15. MPI Forum (2009) MPI: a message-passing interface standard. Version 2.2
16. Plimpton S (1995) Fast parallel algorithms for short-range molecular dynamics. Academic Press Professional. J Comput Phys 117:1–19
17. Reussner R, Träff J, Hunzelmann G (2000) A benchmark for MPI derived datatypes. In: Recent advances in parallel virtual machine and message passing interface (EuroPVM/MPI'00), LNCS, vol 1908. Springer, New York, pp 10–17
18. Schneider T, Gerstenberger R, Hoefler T (2012) Micro-applications for communication data access patterns and MPI datatypes. In: Recent advances in the message passing interface (EuroMPI'12), LNCS, vol 7490. Springer, New York, pp 121–131
19. Skamarock WC, Klemp JB (2008) A time-split nonhydrostatic atmospheric model for weather research and forecasting applications. Academic Press Professional. J Comput Phys 227:3465–3485
20. Träff J, Hempel R, Ritzdorf H, Zimmermann F (1999) Flattening on the fly: Efficient handling of MPI derived datatypes. In: Recent advances in parallel virtual machine and message passing interface (EuroPVM/MPI'99), LNCS, vol 1697. Springer, New York, pp 109–116
21. van der Wijngaart RF, Wong P (2002) NAS parallel benchmarks version 2.4. Technical report, NAS Technical, Report NAS-02-007
22. Wu J, Wyckoff P, Panda D (2004) High performance implementation of MPI derived datatype communication over InfiniBand. In: Proceedings of the international parallel and distributed processing symposium (IPDPS'04). IEEE Computer Society