Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Practical, distributed, low overhead algorithms for irregular gather and scatter collectives[☆,☆☆]

Jesper Larsson Träff

TU Wien, Faculty of Informatics, Research Group Parallel Computing, Favoritenstrasse 16/191-4, Vienna 1040, Austria

ARTICLE INFO

Article history:
Received 29 November 2017
Revised 27 February 2018
Accepted 9 April 2018
Available online 11 April 2018

Keywords:
MPI (Message-Passing Interface)
Collective operations
Irregular collectives
Gather and scatter collectives
Performance guidelines

ABSTRACT

We present new, simple, fully distributed, practical algorithms with linear time communication costs for irregular gather and scatter operations in which processors contribute or consume possibly different amounts of data to or from a chosen root processor. The algorithms consist of a preprocessing phase in which a data distribution aware binomial tree is built, and a communication phase in which the data are gathered to or scattered from the root. In a homogeneous, linear transmission cost model with start-up latency α and cost per unit β, the preprocessing phase takes $2\lceil\log_2 p\rceil$ communication rounds, in each of which only constant sized information is transmitted, and the communication phase another $\lceil\log_2 p\rceil\alpha + \beta\sum_{i\neq r} m_i$ time units, p being the number of processors, m_i the amount of data for processor i, $0 \leq i < p$, and processor r, $0 \leq r < p$, a root processor determined by the algorithm. With a fixed, externally given root processor r, there is an additive time penalty of at most $\beta(M_{d'} - m_{r_{d'}} - \sum_{0\leq j<d'} M_j)$ time units for some $d' < \lceil\log_2 p\rceil$, where each M_j is the total amount of data in a tree of 2^j different processors with roots r_j as constructed by the algorithm. This worst-case time penalty is much less than $\beta\Sigma_{i\neq r} m_i$. The algorithms have attractive properties for implementing the operations for MPI (the Message-Passing Interface). Standard algorithms using fixed, problem oblivious trees take time either $\lceil\log_2 p\rceil(\alpha + \beta\sum_{i\neq r} m_i)$ in the worst case, or $(p-1)\alpha + \sum_{i\neq r}\beta m_i$. We have used the new algorithms to give prototype implementations for the `MPI_Gatherv` and `MPI_Scatterv` collectives of MPI, and present benchmark results from a small and a medium-large InfiniBand cluster. In order to structure the experimental evaluation we formulate new performance guidelines for irregular collectives that can be used to assess the performance in relation to the corresponding regular collectives. We show that the new algorithms can fulfill these performance expectations within a large margin, and that standard implementations do not.

1. Introduction

Gather and scatter operations are important collective operations for collecting and distributing data among processors in a parallel system with some chosen (and known) root processor, e.g., row-column gather-scatter in linear algebra algo-

☆ This work was supported by the Austrian FWF project "Verifying self-consistent MPI performance guidelines" (P25530). The computational results presented have been achieved in part using the Vienna Scientific Cluster (VSC).

☆☆ The paper is a revised, extended version of the paper "Practical, Linear-time, Fully Distributed Algorithms for Irregular Gather and Scatter" presented at the 24rd European MPI Users' Group Meeting (EuroMPI/USA), Argonne, USA, September 2016 [1].

E-mail address: traff@par.tuwien.ac.at

https://doi.org/10.1016/j.parco.2018.04.003
0167-8191/© 2018 Elsevier B.V. All rights reserved.

rithms. The problems come in two flavors, namely a *regular* (or homogeneous) variant in which all processors contribute or consume data blocks of the same size, and an *irregular* (or inhomogeneous) variant in which the data blocks may have different sizes. For the irregular variant, the root processor may or may not know the sizes of the blocks of data to be distributed to or collected from the non-root processors. While good algorithms and implementations exist for different types of systems for the regular problems, the irregular problems have been much less studied and often only trivial algorithms with less than optimal performance (for small to medium block sizes) implemented. In this paper, we present new, simple algorithms for the irregular gather and scatter problems with many desirable properties for practical implementation, and show experimentally with implementations for and in MPI (the Message-Passing Interface) [2] that they can perform much better and much more consistently than commonly used algorithms and implementations.

Gather and scatter operations are included as collective operations in MPI in both variants [2, Chapter 5]. For the regular operations `MPI_Gather` and `MPI_Scatter`, usually fixed (binomial) trees are used (in a hierarchical fashion) for short to medium sized blocks, while large blocks are sent or received directly from or to the root process. Since the common block size is known to all processes, the MPI processes can consistently and without any extra communication decide which algorithm to use. Standard algorithms are surveyed by Chan et al. [3], and analyzed under a homogeneous, linear transmission cost model where they lead to optimal, linear bandwidth, and optimal number of communication rounds (binomial trees). Similar results for different communication networks were presented early by Saad and Schultz [4]. For the irregular `MPI_Gatherv` and `MPI_Scatterv` operations the situation is different. Fixed (block size oblivious) trees of logarithmic depth may lead to a large block being sent a logarithmic number of times, and letting the non-root processes send or receive directly from the root entails a linear number of communication start-ups which might be too expensive when the non-root blocks are small. Furthermore, consistently deciding which of the possible algorithms to use is difficult unless all processes have (full) information on the total sizes of the data blocks, which is in general not the case for the irregular MPI collectives. Current MPI libraries, nevertheless, seem to use variations of these algorithms. Träff [5] gave algorithms specifically for MPI that rely on the global information on block sizes available at the root process and use sorting to construct good trees in a top-down fashion. These algorithms may well be too expensive when non-root blocks are small. Variants of these algorithms were discussed and benchmarked by Dichev et al. [6]. Regular gather-scatter problems for heterogeneous multiprocessors where communication links may have different capabilities have been studied in several papers, e.g., [7,8]. These algorithms also mostly rely on global knowledge (by one process) and sorting by the transmission times between processes to construct good communication schedules, but could be adopted to irregular gather-scatter problems. Boxer and Miller [9] study the regular gather-scatter problems on the coarse grained multiprocessor (CGM) and concentrate on the problem of finding good spanning trees for the machine in case. For hypercubes, compound scatter-gather computations are studied more precisely by Charles and Fraigniaud [10] who derive pipelined schedules for the regular gather and scatter problems. Simple algorithms for the regular problems in an asynchronous communication model that accounts for delays and permits overlap were presented by Shibusawa et al. [11]. Bhatt et al. [12] study the irregular gather and scatter problems in tree networks, and derive (nearly) optimal schedules for arbitrary trees. This situation is somewhat orthogonal to the usual objective of finding both a good spanning tree and a corresponding schedule. The algorithms require full knowledge of the message sequences to be scattered and gathered.

In the following we present new, simple algorithms for the irregular gather and scatter problems with a number of desirable properties. For the analysis, we assume a homogeneous, fully connected network with 1-ported, bidirectional (telephone-like) communication. We let p denote the number of processors which are numbered consecutively from 0 to $p-1$. The cost of transmitting a message of m units between any two processors is linear and modeled as $\alpha+\beta m$, where α is a communication start-up latency, and β the transmission time per unit. Both processors are involved throughout the message transmission, and can do nothing else during that time. The system is asynchronous, and a processor can start the next transmission as soon as it has finished communication and select from which other processor to receive the next message. In this setting the information dissemination lower bound of $\lceil\log_2 p\rceil$ rounds for synchronized systems does not apply [13], and optimal trees may have root degree less than $\lceil\log_2 p\rceil$ [14]. In the irregular gather and scatter problems, each processor i, $0\leq i<p$, has a block of data of size m_i with $m_i\geq 0$ that it either wants to contribute to (gather) or consume from (scatter) some root processor r, $0\leq r<p$. The root r is usually a given processor, and this r is known to all other processors. At the root, blocks are stored in processor order, that is $m_0, m_1, m_2, \ldots, m_r, \ldots, m_{p-1}$ (we assume that the root also has a block m_r for itself that does not have to be transmitted but possibly copied from one buffer to another). Any consecutive sequence of blocks can be sent or received together as a single message. Our algorithms do not assume that the root knows the size of all p data blocks, although the `MPI_Gatherv` and `MPI_Scatterv` operations do make this assumption and require this to be the case.

Our algorithms construct spanning trees of logarithmic depth, and need only the optimal $d=\lceil\log_2 p\rceil$ number of communication rounds for the tree construction, each round consisting of at most two communication steps. Trees are constructed in a distributed manner, with each processor working only from gradually accumulated information, with no dependence on global information (e.g., from the root) on the sizes of other data blocks. For the gathering or scattering of the data blocks, another at most $\lceil\log_2 p\rceil$ communication rounds are needed. In the constructed trees, the time for the root to gather or scatter all data blocks from or to the non-root processors is linear, namely $\lceil\log_2 p\rceil\alpha+\beta\sum_{0\leq i<p, i\neq r} m_i$, with an additive time penalty of at most $\beta(M_{d'}-m_{r_{d'}}-\sum_{0\leq j<d'} M_j)$ time units for some $d'<d$ where each M_j is the total amount of data in a tree of 2^j different processors as constructed by the algorithm for the case when the root is a fixed, externally given pro-

cessor (as is the case in MPI_Gatherv and MPI_Scatterv). This worst-case time penalty is (much) less than $\beta\Sigma_{i\neq r}m_i$ which would be incurred if data were transmitted between some good root determined by the algorithm and the given root processor r. In contrast, for any fixed, block-size oblivious binomial tree it is easy to construct a worst case problem instance taking $\lceil\log_2 p\rceil(\alpha+\beta\sum_{0\leq i<p,i\neq r}m_i)$ time steps, namely by choosing $m_i=0$ (or some small size, in case blocks of size 0 are not transmitted) for all processors except one being farthest away from the root. At all processors, blocks are always sent and received in processor order: Any receive operation receives a message consisting of blocks $m_k, m_{k+1}, \ldots, m_{k+l}$. Therefore, no potentially costly, local reordering of blocks in message buffers is necessary at any stage of our algorithm.

We have implemented our algorithms to support the MPI_Gatherv and MPI_Scatterv operations, and evaluated them with different block size distributions on a small InfiniBand cluster under three different MPI libraries, and a medium-large InfiniBand cluster under the vendor (Intel) MPI library.[1] In order to structure the comparison against the native MPI library implementations we formulate expectations on the relative performance as new, self-consistent performance guidelines [15,16]. We can show that the new algorithms can in many situations significantly outperform the native MPI library, and overall much better fulfill the formalized performance expectations.

2. Problem and algorithm

We now present the algorithm for the irregular gather problem; the scatter algorithm is analogous. Each of the p processors has a data block of m_i units that it needs to contribute to some root processor r, $0\leq r<p$. We organize the p processors in a $\lceil\log_2 p\rceil$-dimensional (incomplete), ordered hypercube which we use as a design vehicle, but communication can be between processors that are not adjacent in the hypercube. We let H_d, $0\leq d\leq\lceil\log_2 p\rceil$ denote a d-dimensional (incomplete) hypercube consisting of (at most) 2^d processors. We say that the hypercube H_d is *ordered* if the processors belonging to H_d form a consecutive range $[a2^d,\ldots,a2^d+2^d-1]=[a2^d,\ldots,(a+1)2^d-1]$ for $a\in\{0,\ldots,\lceil p/2^d\rceil-1\}$. The ordered hypercube H_{d+1} consisting of processors $[a2^{d+1},\ldots,(a+1)2^{d+1}-1]$ is built from two *adjacent*, ordered hypercubes H_d with processors $[2a2^d,\ldots,(2a+1)2^d-1]$ and $[(2a+1)2^d,\ldots,(2a+2)2^d-1]$. If p is not a power of two, the last H_d hypercube consists of the processors $[(\lceil p/2^d\rceil-1)2^d,\ldots,p-1]$.

By an *ordered hypercube gather algorithm* for H_d we mean an algorithm for H_d in which a processor in one of the subcubes H_{d-1} which has gathered all data from the processors of this subcube sends all its data to a processor in the other, adjacent subcube H_{d-1} which similarly has already gathered all data from that subcube. This processor will now have gathered all data in the hypercube H_d and will become the root processor of H_d. Note that this may require communication along an edge that is not a hypercube edge, but of course does belong to the fully connected network.

Lemma 1. *For any H_d, there exists an ordered hypercube gather algorithm that gathers the data to some root processor r in H_d in time $d\alpha+\beta\sum_{i\in H_d,i\neq r}m_i$.*

Proof. The claim follows by induction on d. For H_0 the sole processor $r\in H_0$ already has the data m_0 and there is no further cost. Let H'_{d-1} and H''_{d-1} be the two adjacent subcubes of H_d for $d>0$. By the induction hypothesis there is a processor r' of H'_{d-1} that has gathered all data of H'_{d-1} in $t'=(d-1)\alpha+\beta\sum_{i\in H'_{d-1},i\neq r'}m_i$ time steps, and a processor r'' that has gathered all data of H''_{d-1} in $t''=(d-1)\alpha+\beta\sum_{i\in H''_{d-1},i\neq r''}m_i$ time steps. Of the two root processors r' and r'', the one with the smaller gather time (with ties broken in favor of the hypercube with the smallest amount of data) sends its data to the other root processor. Say, r' is the root with $t'\leq t''$. Processor r' sends a message of $\sum_{i\in H'_{d-1}}m_i$ units to root r'' which takes $\alpha+\beta\sum_{i\in H'_{d-1}}m_i$ time steps. Adding to the time t'' already taken by the slower r'' to gather the data from H''_{d-1} gives $(d-1)\alpha+\beta\sum_{i\in H''_{d-1},i\neq r''}m_i+\alpha+\beta\sum_{i\in H'_{d-1}}m_i=d\alpha+\beta\sum_{i\in H_d,i\neq r}m_i$ as claimed. The root r'' of H''_{d-1} becomes the root r of H_d. $\square$

Since roots with smaller gather times send to roots with larger gather times, communication can readily take place with no delay for the sending gather root processor to become ready. Since subcubes are ordered, the data blocks received at a new root can easily be kept in consecutive order. Note that for the gather times of the two roots r' and r'', $t'\leq t''$ if and only if $\sum_{i\in H'_{d-1},i\neq r'}m_i\leq\sum_{i\in H''_{d-1},i\neq r''}m_i$, so that the shape of the constructed gather tree depends only on the block sizes and not on the relative magnitudes of α and β. For each H_d hypercube with root r, $\sum_{i\in H_d,i\neq r}m_i$ is an estimate of the time to construct H_d.

Lemma 2. *For any arbitrarily given root processor $r\in H_d$, there is an ordered hypercube algorithm that gathers all data in H_d to r in $d\alpha+\beta\sum_{i\in H_d,i\neq r}m_i$ time units with an additive time penalty of at most $\beta(M_{d'}-m_{r_{d'}}-\sum_{0\leq j<d'}M_j)$ for some d', $d'<d$. The root processor gathers data from the roots in a sequence of ordered hypercubes $H_0, H_1, \ldots, H_{d-1}$, each with a total amount of data M_j, and $H_{d'}$ is the last such hypercube for which waiting time is incurred.*

Proof. The construction of Lemma 1 is modified such that data are always sent to processor r if either $r'=r$ or $r''=r$. The given root processor r will therefore receive blocks from $d-1$ linear gather time subcubes $H_0, H_1, \ldots, H_{d-1}$. The amount of

[1] The implementations used for evaluation are available at www.par.tuwien.ac.at/Downloads/TUWMPI/tuwgatherv.c .

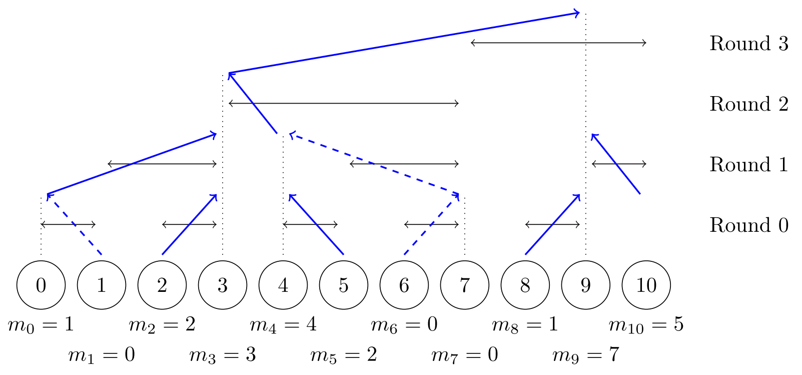

Fig. 1. A linear-time, ordered gather tree for $p = 11$ processors and root $r = 9$ with the indicated block sizes m_i as constructed by the algorithm of Lemma 3. Thick (blue), upward arrows are the gather tree edges with dotted arrows indicating data sizes of zero with no actual communication. Thin (black), horizontal double arrows indicate the exchange between fixed roots as needed to construct the ordered gather tree and illustrated in Fig. 2. Vertical, dotted lines indicate the rounds in which processors are active sending or receiving data blocks. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article).

data, and the time needed to gather the data in these d hypercubes is unrelated and may differ. Let $M_j = \sum_{i\in H_j} m_i$ be the amount of data in hypercube H_j with root processor r_j. If the time needed to gather the data in some $H_{d'}$ to $r_{d'}$, namely $\alpha d' + \beta(M_{d'} - m_{r_{d'}})$, is larger than the time needed to gather the data from the previous hypercubes $H_0, H_1, \ldots H_{d'-1}$ to r, the root processor is delayed until the data gather in $H_{d'}$ has completed. This delay is at most $\alpha d' + \beta(M_{d'} - m_{r_{d'}}) - (\alpha d' + \beta(\sum_{j<d'} M_j) = \beta(M_{d'} - m_{r_{d'}} - \sum_{j<d'} M_j)$. Let $H_{d'}$ be the last hypercube in the sequence incurring such a delay. The total time to gather all data to the root r is $d\alpha + \beta\sum_{i\in H_d, i\neq r} m_i$ plus the penalty of $\beta(M_{d'} - m_{r_{d'}} - \sum_{j<d'} M_j)$. □

The resulting construction is easy to implement, and better than first gathering to the linear time root determined by Lemma 1 and then sending to the externally given root r which would incur an extra communication round to send the complete data $\sum_{i\in H_d} m_i$, effectively loosing half of the communication bandwidth. It is important to notice that we assume that for each hypercube H_d with root r there is no cost associated with the root block m_r. We return to this point in Section 3.

The communication structure of an ordered hypercube gather algorithm is a binomial tree with a particular numbering of the tree roots determined by the data block sizes m_i, $0 \leq i < p$. An example is shown in Fig. 1. This tree can be constructed efficiently as shown by the next lemma which is illustrated in Fig. 2.

Lemma 3. *For any H_d, a linear-time gather communication tree can be constructed in d communication rounds, each comprising at most two send and receive operations.*

Proof. The communication tree is constructed iteratively, maintaining the following invariant. For each H_d, a predetermined, fixed root that can readily be computed by any processor is chosen. Each H_d will maintain a gather root r that will gather the data from H_d as per Lemma 1. The fixed root and the gather root are not necessarily distinct processors. Both the fixed and the gather root processors know that they have this role and which processor has the other role, and each knows the total amount of data in H_d. When the hypercube H_{d+1} is formed from H'_d and H''_d, the fixed root of H'_d knows which processor is the fixed root of H''_d and vice versa. The gather roots do not know the gather root of the other subcube.

For all H_0 subcubes the invariant holds with fixed and gather root being the sole processor in each H_0. To maintain the invariant for H_{d+1}, the two fixed roots of the H_d subcubes exchange information on their estimated gather time, the size of the root data blocks, and the identity of the gather root processors. Both fixed roots can now determine which gather root will be the gather root of H_{d+1}, namely the gather root of the subcube with the largest gather time estimate $\sum_{i\in H_d, i\neq r} m_i$ (with ties broken arbitrarily, but consistently). The first time a fixed root of some H_d by the exchange determines that it will become a gather root of H_{d+1}, this new gather root knows that it is a gather root. To maintain the invariant for the following iterations, the gather root of H_d which does not know whether it will be the gather root of H_{d+1} (unless fixed and gather roots are the same processor), receives information on the gather root in H_{d+1} from its fixed root in H_d which per invariant knows the identity of the gather root in H_d. By exchanging both the gather times $\sum_{i\in H_d, i\neq r} m_i$ and the sizes of the root data blocks m_r, gather roots can compute the amount of data to be received in each communication round.

The construction takes $\lceil \log_2 p \rceil$ iterations in each of which pairwise exchanges between the fixed roots of adjacent hypercubes take place. After this, at most one transmission between each fixed root and its corresponding gather root is necessary, except for the first communication round where no such transmission is needed. Thus at most $2\lceil \log_2 p \rceil - 1$ de-

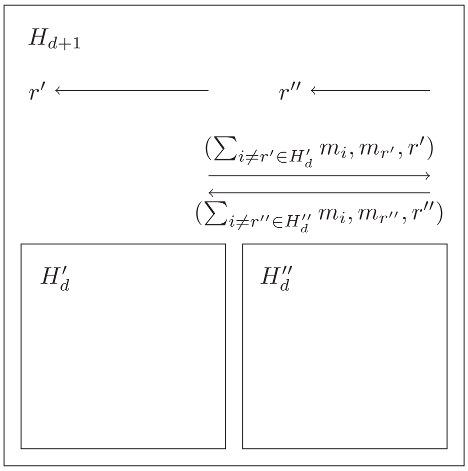
H_{d+1}
r'
r''
$(\sum_{i\neq r'\in H'_d} m_i, m_{r'}, r')$
$(\sum_{i\neq r''\in H''_d} m_i, m_{r''}, r'')$
H'_d
H''_d

Fig. 2. An iteration of the algorithm of Lemma 3 showing the communication necessary to join two adjacent, ordered hypercubes H'_d and H''_d (small squares) into the larger hypercube H_{d+1} (large square containing the two small squares). The fixed roots first exchange information on the gather times, the sizes of the root data blocks, and the identity of the gather roots in the respective subcubes. In the next step, the gather roots r' and r'' receive this information from their fixed roots, so that they can consistently determine which will be the gather root for H_{d+1}.

pendent communication operations are required. All information exchanged is of constant size, consisting of the gather time estimate, the size of the root data block, and the identity of the gather root. □

As fixed root for a subcube H_d consisting of processors $[a2^d, \dots, (a+1)2^d-1]$ we can choose, e.g., the last processor $i=(a+1)2^d-1$. For the fixed root of this H_d to find the fixed root of its adjacent subcube, the d'th bit of i has to be flipped. If the d'th bit is a 1, the processor will survive as the fixed root of H_{d+1}. If the number of processors p is not a power of two, only the last processor $p-1$ has to be specially treated. If a fixed root in iteration d, by flipping bit d, determines that its partner fixed root is larger than $p-1$ it instead chooses $p-1$ as fixed root in its adjacent, incomplete hypercube. In this iteration, processor $p-1$ must be prepared to act as fixed root. In iteration d, if processor $p-1$ has bit d set, it knows that some lower numbered fixed root has chosen $p-1$ as fixed root, and this adjacent fixed root is $(\lceil p/2^d\rceil-1)2^d-1$. Otherwise, if bit d is not set, processor $p-1$ has no role in iteration d. Thus, deciding the identity of the fixed roots can always be done in constant time.

Together, these remarks and the three lemmas give the main result.

Theorem 1. *For any number of processors p in a homogeneous, fully connected, linear time cost communication network, the irregular gather problem with root r and block size m_i for processor i, $0\leq i<p$ can be solved in time at most $\lceil\log_2 p\rceil\alpha+\beta\sum_{0\leq i<p,i\neq r} m_i$ after a preprocessing step taking at most $2\lceil\log_2 p\rceil$ communication rounds with an additive time penalty of at most $\beta(M_{d'}-m_{r_{d'}}-\sum_{0\leq j<d'} M_j)$ time units for some $d'<\lceil\log_2 p\rceil$, each M_j being the total amount of data in a tree of 2^j distinct processors with local root r_j.*

The linear time gather trees can likewise be used for the irregular scatter problem with the same time bounds. Also, both tree construction and communication algorithms can be extended to k-ported communication systems, which reduces the number of communication rounds needed from $\lceil\log_2 p\rceil$ to $\lceil\log_{k+1} p\rceil$. It is perhaps worth pointing out that the constructions also provide ordered communication trees for the regular gather and scatter (as well as for reduction-to-root) operations with the optimal $\lceil\log_2 p\rceil$ number of communication rounds. If all processors know the common root r, tree construction can be done without any actual, extra communication.

3. MPI implementations

We have implemented the irregular gather and scatter algorithms described in the previous section in MPI with the same interface as the corresponding MPI operations, and can thus readily compare our `TUW_Gatherv` and `TUW_Scatterv` operations to different MPI library implementations.[2] We use the algorithm of Lemma 3 to construct communication trees which we (for gather) represent as a sequence of receive operations followed by a send operation at each process. Where appropriate, we use non-blocking sends and receives to better absorb delays by some MPI processes finishing late. For non-root processes, intermediate buffers gather (scatter) data from (to) the processes' children. Since the sizes of all received (sent)

[2] The implementations are available at www.par.tuwien.ac.at/Downloads/TUWMPI/tuwgatherv.c, and implement the constructions from both Lemma 1 and Lemma 4. There are likewise switches for using either non-blocking or blocking communication for the data blocks.

data are known after the tree construction, and since all children send (receive) rank ordered data blocks, it is straightforward to keep blocks stored in intermediate buffers in rank order. Each of the processes copies its own block into the intermediate communication buffer at the appropriate offset; blocks received from processes with smaller ranks are put before, and blocks received from processes with larger ranks after this offset. The copy of the process' own block to/from the intermediate buffer incurs a cost proportional to m_i not accounted for in Lemma 1. This explicit copy could be avoided by using an indexed derived MPI datatype to handle communication of this local block. Alternatively, Lemma 4 below accounts for an explicit copy of the processes' own, local block under the pessimistic assumption that a local copy takes at most βm_i time units.

Lemma 4. *Assuming that communication is always out of a dedicated communication buffer, and that each processor has to copy it's own block m_i into such a communication buffer at cost βm_i, there exists for any H_d an ordered hypercube gather algorithm that gathers the data to some root processor r in H_d in time $d\alpha + \beta \sum_{i \in H_d} m_i$.*

Proof. The claim follows by induction on d. For H_0 the sole processor $r \in H_0$ copies its data m_0 into the communication buffer at cost βm_0 as claimed. Let H'_{d-1} and H''_{d-1} be the two subcubes of H_d. By the induction hypothesis there is a processor r' of H'_{d-1} that has gathered all data of H'_{d-1} in $t' = (d-1)\alpha + \beta \sum_{i \in H'_{d-1}} m_i$ time steps, and a processor r'' that has gathered all data of H''_{d-1} in $t'' = (d-1)\alpha + \beta \sum_{i \in H''_{d-1}} m_i$ time steps. Of the two root processors r' and r'', the one with the smaller gather time (with ties broken arbitrarily but consistently) sends its data to the other root processor. Say, r' is the root with $t' \le t''$. Processor r' sends a message of $\sum_{i \in H'_{d-1}} m_i$ units to root r'' which takes $\alpha + \beta \sum_{i \in H'_{d-1}} m_i$ time steps. Adding to the time t'' already taken by the slower r'' to gather the data from H''_{d-1} gives $(d-1)\alpha + \beta \sum_{i \in H''_{d-1}} m_i + \alpha + \beta \sum_{i \in H'_{d-1}} m_i = d\alpha + \beta \sum_{i \in H_d} m_i$ as claimed. The root r'' of H''_{d-1} becomes the root r of H_d. □

It is worth noting that under the copy cost assumption in Lemma 4, the cost of gathering in a hypercube H_d is proportional to the sum of the blocks in H_d. To build the gather tree as in Lemma 3, only the total sizes of hypercubes need to be exchanged in addition to the gather roots. Similar lemmas hold for the scatter operations, and the implementations of `TUW_Gatherv` and `TUW_Scatterv` are entirely similar. There is a difference between the two cases though that will affect actual performance. Both gather and scatter trees are constructed bottom-up as described in Lemma 3. In the gather operation data blocks are likewise communicated upwards toward the root, and the tree construction can therefore be hidden behind the actual communication with an effective communication penalty of just one or a few constant-sized communication rounds. In the scatter operation, the root cannot start scattering data blocks downwards before the tree has been constructed, and therefore has to pay the full delay of a logarithmic number of constant-sized rounds before the actual communication can start.

All processes in the `MPI_Gatherv` and `MPI_Scatterv` operations supply an MPI datatype describing the types and structure of their blocks. Since all data blocks must eventually match the datatype supplied by the root process, it is possible to receive and send all intermediate data blocks with a correct MPI derived datatype. To this end, the signature datatype described in [17] can be used. Since blocks can be described by different types with different counts by different processes, it is important that the smallest, repeated unit of the signature type is used as communication datatype. At the root, data blocks are to be stored as described by the list of displacements and block sizes supplied in the root process' `MPI_Gatherv` or `MPI_Scatterv` call. This can be accomplished by constructing a corresponding indexed derived datatype for each of the children describing where the data blocks go. No explicit, intermediate buffering at the root is therefore necessary, and in that sense zero-copy implementations [18,19] of the algorithms are possible. However, zero-copy implementations with MPI derived datatypes may come with a non-negligible data type set-up overhead, making implementations without preferable. If the root displacements describe a contiguous segment of blocks in rank order (as may be the case in applications), no such datatype is necessary, and the blocks can be received directly into their correct positions in the root receive buffer. Our prototype implementation works under this assumption.

Despite the linear gather and scatter times guaranteed by the algorithm, sending large data blocks multiple times through the tree incurs unnecessary, repeated transmission costs. Practical performance might be better if such large blocks would be sent directly to or from the root process. It would be possible to implement graceful degradation behavior [5] by introducing a gather subtree threshold beyond which a subtree in the gather tree shall send its data directly to the root. This is not entirely trivial, and we have not yet implemented this potential improvement.

4. Performance guidelines for irregular collectives

Self-consistent MPI performance guidelines formalize expectations on the performance of given MPI operations by relating them to the performance of other MPI operations implementing the same functionality [16]. If a performance guideline is violated, it gives a constructive hint to the application programmer and the MPI library implementer how the given operation can be improved in the given context [20]. Performance guidelines thus provide sanity checks for MPI library implementations, and can be helpful in structuring experiments [15,21].

In order to use regular collectives correctly, the application programmer must know that all processes supply the same data sizes and each process must know this data size. The irregular collectives have a weaker precondition: It suffices that each process by itself knows its data size with the only requirement that processes that pairwise exchange data must know and supply the same sizes. If an irregular collective is used in a situation where a regular one could have been used instead, we would expect the regular collective to perform better, or at least not worse in that situation. This is captured in the performance guideline for MPI_Gatherv below.

$$\texttt{MPI\_Gather}(m) \preceq \texttt{MPI\_Gatherv}(m) \tag{GL1}$$

Here m is the total amount of data to be gathered at the root process, and the guideline states that in a situation where MPI_Gather can be used ($m_i = m/p$), this should perform at least as well as using instead MPI_Gatherv, all other things (e.g., root process, communicator) being equal in the two sides of the inequality. If the guideline is violated, which can be tested experimentally, there is something wrong with the MPI library, and the user would do better by using MPI_Gatherv instead of MPI_Gather. There are reasons to expect that the guideline is not violated. The MPI_Gather operation is more specific, does not take long argument lists of counts and displacements, and good, tree-based algorithms exist and may have been implemented for this operation [3,22].

A common way of dealing with slightly irregular problems is to transform them into regular ones by padding all buffers up to some common size and solving the problem by a corresponding regular collective operation. The argument for having the specialized, irregular collectives in the MPI specification is that a library can possibly do better than (or at least as good as) this manual solution. Thus, we would like to expect that MPI_Gatherv performs no worse than first agreeing on the common buffer size and then doing the regular collective on this, possibly larger common size. This is expressed in the second, irregular collectives performance guideline.

$$\texttt{MPI\_Gatherv}(m) \preceq \texttt{MPI\_Allreduce}(1) + \texttt{MPI\_Gather}(m') \tag{GL2}$$

where $m' = p \max_{0 \le i < p} m_i$ is the total amount of data to be gathered by the regular MPI_Gather as computed by the MPI_Allreduce operation. Again, if experiments show this guideline violated, there is an immediate hint for the application programmer on how to do better: Use padding. Here we assume that the application programmer can organize the padded buffers such that no copying back and forth between buffers is necessary; this may not always be possible, so the guideline should not be interpreted too strictly but allow some extra slack on the right-hand side upper bound. Nevertheless, it constrains what should be expected of a good implementation of MPI_Gatherv.

The second guideline is particularly interesting for the regular case where $m_i = m/p$. Here it says that the overhead in using MPI_Gatherv compared to MPI_Gather should not be more than a single, constant-sized MPI_Allreduce operation. This may be difficult for MPI libraries to satisfy, but indeed, if it is not, the usefulness of MPI_Gatherv may be questionable.

The two performance guidelines give less trivial performance expectations against which to test our new algorithms instead of only comparing to the MPI_Gatherv and MPI_Gather implementations in some given MPI library. Analogous performance guidelines can be formulated for MPI_Scatterv and extended to the other, irregular MPI collectives.

5. Experiments

Finally, we give an indicative, experimental evaluation of both the TUW_Gatherv and the TUW_Scatterv implementations. We evaluate by comparing to MPI_Gather and MPI_Gatherv, and MPI_Scatter and MPI_Scatterv of common MPI libraries guided by the performance guidelines explained in Section 4. The aim is to explore whether standard MPI libraries fulfill reasonable performance guidelines for collective operations, and, if not, whether the TUW_Gatherv and TUW_Scatterv implementations given here fare better. The native, library implementations of MPI_Gather and MPI_Scatter are the baselines for the performance guidelines; it is not the purpose to investigate potential performance problems with these operations.

We test our algorithm on problems of varying degrees of irregularity. Let b, $b > 0$ be a chosen, average block size (in some unit, here MPI_INT). We have p MPI processes, and use as fixed gather root $r = \lfloor p/2 \rfloor$. Our problems are as follows with names indicating how block sizes are chosen for the processes.

Same: For process i, $m_i = b$, all blocks have the same size.
Random: Each m_i is chosen uniformly at random in the range [1, $2b$].
Bucket: Each m_i is $b/2$ plus a contribution chosen uniformly at random in the range [1, b].
Spikes: Each m_i is either ρb, $\rho > 1$ or 1, chosen randomly with probability $1/\rho$ for each process i.
Increasing: For process i, $m_i = \lfloor \frac{2b(i+1)}{p} \rfloor$
Decreasing: For process i, $m_i = \lfloor \frac{2b(p-i)}{p} \rfloor + 1$
Alternating: For even numbered processes, $m_i = b + \lfloor b/2 \rfloor$, for odd numbered processes $m_i = b - \lfloor b/2 \rfloor$.
Two blocks: All $m_i = 0$, except $m_0 = b$ and $m_{p-1} = b$.

These problem types, except for the last, specifically always have $m_i > 0$. This choice ensures that an implementation cannot take advantage of not having to send empty blocks. We perform a series of (weak scaling) experiments with

$b = 1, 10, \ldots, 10\,000$; the total problem size in each case is $m = \sum_{0 \le i < p} m_i$ and increase linearly with p (except for the **two blocks** problems). The problem block size structures are chosen such that all problems (except for the **two blocks** problems) have roughly the same size for each value of b. For comparison with MPI_Gather and MPI_Scatter, and for the padding performance Guideline (GL2), the padded block size is $\max_{0 \le i < p} m_i$ and the total size $m' = p \max_{0 \le i < p} m_i$. For the **spikes** problems, we have taken $\rho = 5$. For each problem, we measure the time for the MPI library native MPI_Gather and MPI_Scatter operations on the padded problem instance, the time for the right-hand side of Guideline (GL2) (MPI_Allreduce followed by MPI_Gather or MPI_Scatter on padded problem), the time for the MPI library native MPI_Gatherv and MPI_Scatterv operations, and finally the time for our TUW_Gatherv and TUW_Scatterv implementations. Performance differences between library native MPI_Gatherv and MPI_Scatterv, and the new TUW_Gatherv and TUW_Scatterv implementations can thus be studied, and violations to Guideline (GL2) assessed by comparing the gather and scatter running times against the measured performance guideline right hand side. The irregular gather and scatter trees are constructed as summarized in Theorem 1, that is assuming no cost for process local block copying. Experiments showed no essential performance difference between trees constructed under the cost assumptions in Lemmas 1 and 4. We emphasize that the TUW_Gatherv and TUW_Scatterv implementations include both tree construction as well as the actual gather/scatter data block communication, so that all comparisons are of same functionality against same functionality.

In our experiments we perform 75 time measurements of each of the collective operations with 10 initial, not timed, warmup calls. In the plots shown here, we report the minimum times (the fastest completion time seen over the 75 repetitions), which are sometimes argued to be stable and reproducible [23]. Average completion times without any outlier removal can be found in the accompanying technical report [24]. Before each measurement, MPI processes are synchronized with the native MPI_Barrier operation, and the running time of a measurement is the time for the slowest process to complete.

Our first test system is a small InfiniBand cluster with 36 nodes each consisting of two 8-core AMD Opteron 6134 processors running at 2.3 GHz. The interconnect is a QDR InfiniBand MT26428. We have tried the gather and scatter implementations with three different MPI libraries, namely NEC MPI-1.3.1, MVAPICH2-2.2 and OpenMPI-2.0.1 using the gcc 4.4.7 compiler with -O3 optimization. We ran experiments with $p = 35 \times 16 = 560$ MPI processes, and present the results in Figs. 3–8. The plotted minimum completion times over the 75 repetitions are in microseconds (µs).

The results show the three library implementations of the MPI_Gather and MPI_Gatherv, MPI_Scatter and MPI_Scatterv operations to differ considerably in quality and behavior. For MPI_Gather, this can best be seen for the **same** problem type, where the NEC MPI minimum time is about 10,500 µs (Fig. 3), the MVAPICH minimum time about 31,000 µs (Fig. 5), and the Open MPI minimum time about 13,800 µs (Fig. 7) for the largest $b = 10\,000$ problem instance. For MPI_Scatter, the corresponding figures are 11,400 µs (Fig. 4), 9400 µs (Fig. 6), and 16,500 µs (Fig. 8), respectively. Also for the smaller to medium problem sizes, the differences can be considerable, more than a factor of two, with the NEC MPI library being the fastest for both MPI_Gather and MPI_Scatter. For the three libraries, it seems that MPI_Gatherv is implemented with a trivial, linear communication round algorithm compared to MPI_Gather; this makes MPI_Gatherv a very expensive operation for small problem sizes and leads to severe violations of Guideline (GL2), with the exception of the **two blocks** problems where MPI_Gatherv is faster than TUW_Gatherv confirming the assumption that a linear implementation is used. For MPI_Scatterv, the NEC MPI library apparently uses a different, tree-based algorithm with similar performance to MPI_Scatter (Fig. 4), whereas the two other libraries again uses a trivial implementation with a very high penalty for small problems.

In almost all cases, TUW_Gatherv performs considerably better than MPI_Gatherv for the smaller problems and in many cases even for the whole problem size range, often by an order of magnitude or more. For the Open MPI library, there is a conspicuous exception for the **same** and **decreasing** problems for $b = 1000$, where MPI_Gatherv is significantly better than TUW_Gatherv; the reason for this (reproducible) behavior has not been investigated (Fig. 7). For the smallest problems with $b = 1$, the difference in performance between TUW_Gatherv and MPI_Gatherv is between a factor of 4 (for the Open MPI library) and 50 (for the NEC MPI library) for the **same** problem type. The same holds for TUW_Scatterv versus MPI_Scatterv, except for the NEC MPI library which uses a better implementation on par with or better than the TUW_Scatterv implementation (Figs. 3 and 4).

Over the different problem types (excluding the **two blocks** problems), running times for TUW_Gatherv and TUW_Scatterv are of the same order, as Theorem 1 predicts, with some interesting exceptions. For $b = 10\,000$, **spikes** is faster, and **decreasing** slower than the average time for the NEC MPI (Figs. 3 and 4) and the MVAPICH (Figs. 5 and 6) libraries; for the Open MPI library (Figs. 7 and 8), the **increasing** problem type is faster, and **decreasing** type slower. The overhead for the tree construction in TUW_Gatherv and TUW_Scatterv can be assessed with the **two blocks** problem for $b = 1$. For all three libraries it is roughly the same for the two cases, being around 20 µs for the NEC MPI library, 33 µs for MVAPICH, and 60 µs for Open MPI.

The trivial performance Guideline (GL1) can be checked by the results for the **same** problems. For the MVAPICH library it is violated quite severely for MPI_Gather with TUW_Gatherv being up to a factor of two faster (Fig. 5), and to some extent also for MPI_Scatter (Fig. 6). Also Open MPI violates the guideline for $b = 100$ and $b = 1000$, indicating poor MPI_Gather and MPI_Scatter implementations for these libraries (Figs. 7 and 8). The NEC MPI library does not violate this guideline.

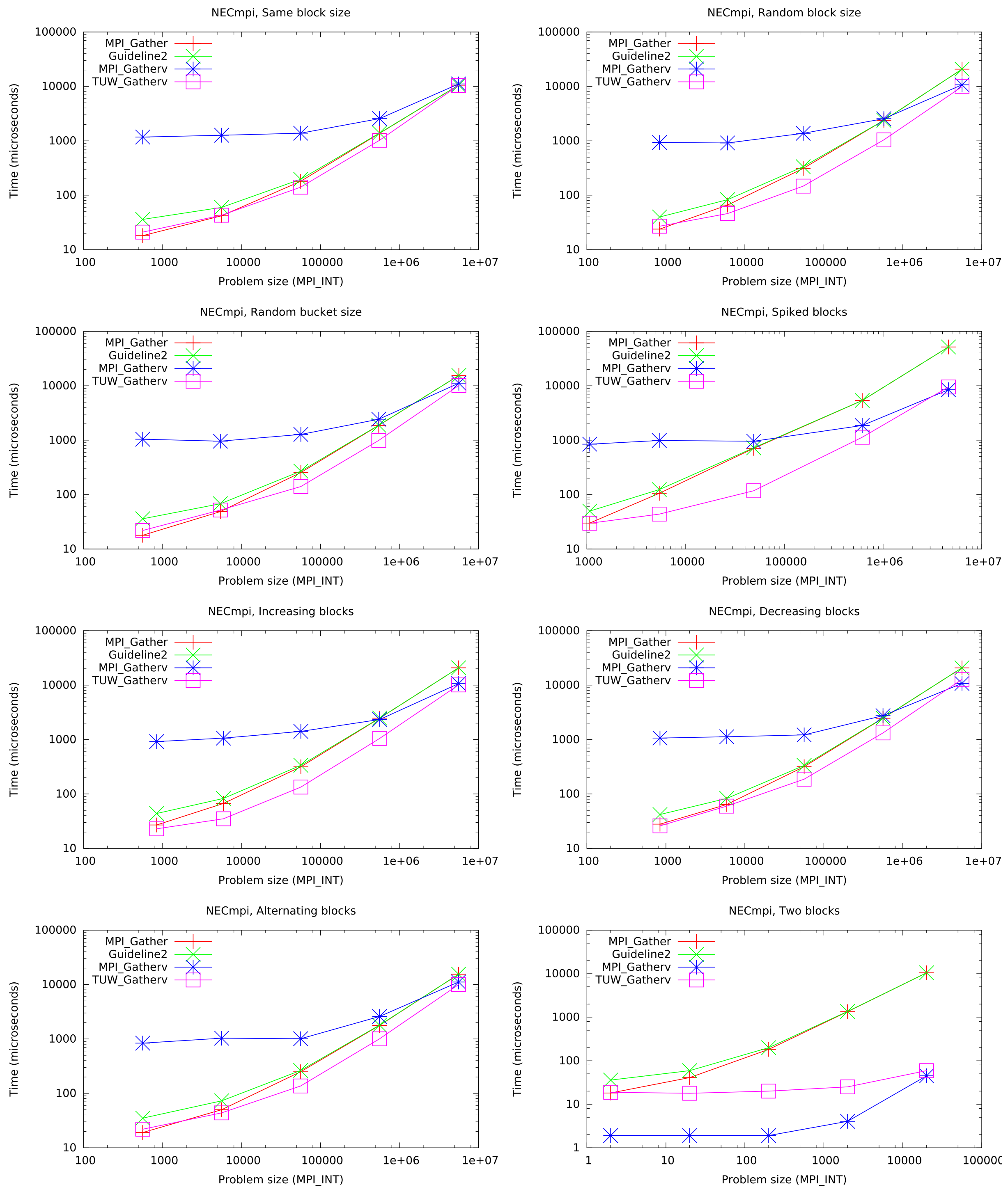

Fig. 3. Results for NEC MPI with $p = 35 \times 16 = 560$ processes for (padded) `MPI_Gather`, right-hand side of Guideline (GL2), `MPI_Gatherv` and `TUW_Gatherv`. Minimum completion times over 75 repetitions times are in microseconds (μs), problem sizes in number of `MPI_INT` elements. Fulfillment of Guideline (GL2) is witnessed by the completion time of `TUW_Gatherv` (or `MPI_Gatherv`) being smaller than the guideline time.

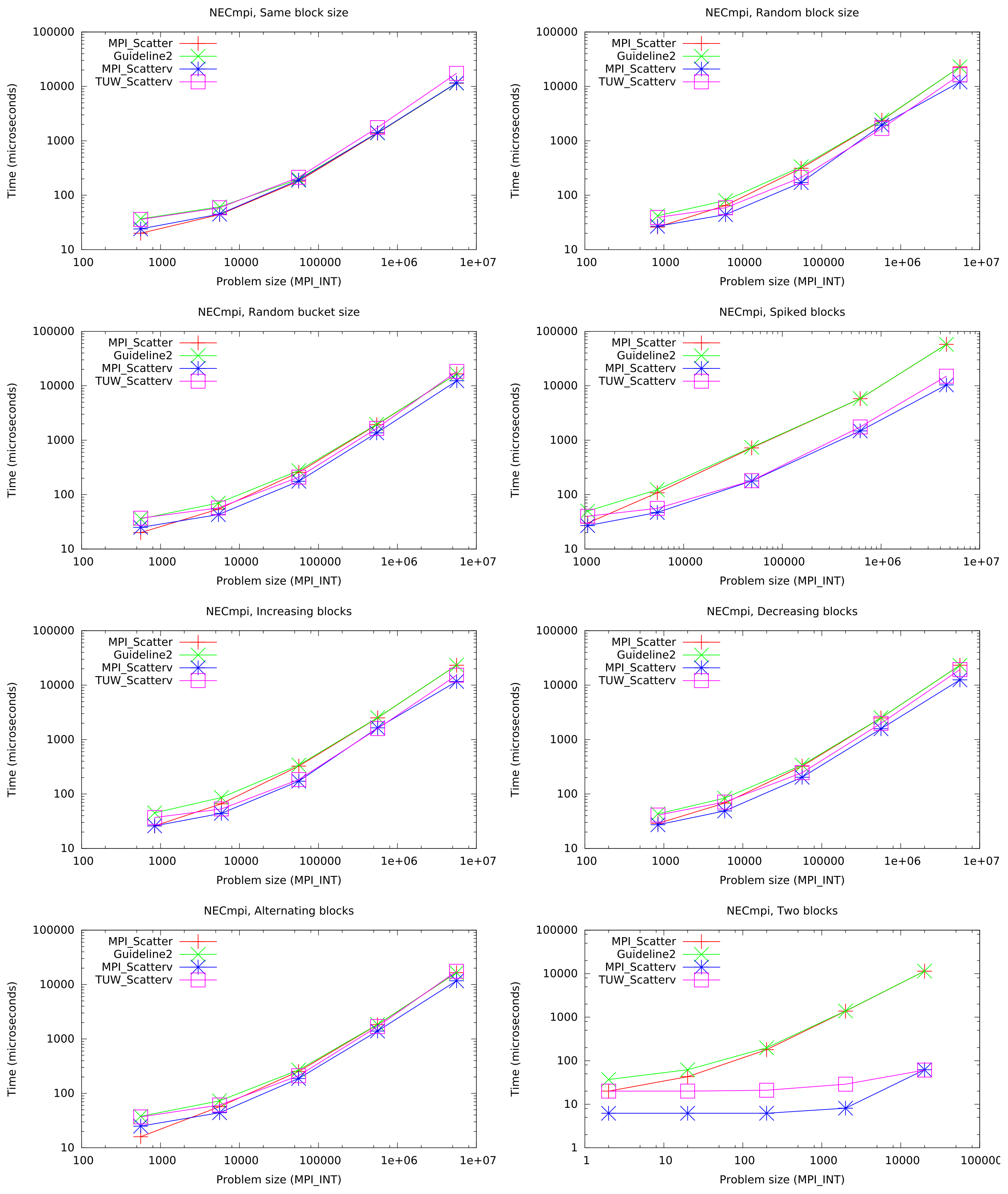

Fig. 4. Results for NEC MPI with $p = 35 \times 16 = 560$ processes for (padded) `MPI_Scatter`, right-hand side of Guideline (GL2), `MPI_Scatterv` and `TUW_Scatterv`. Minimum completion times over 75 repetitions times are in microseconds (µs), problem sizes in number of `MPI_INT` elements. Fulfillment of Guideline (GL2) is witnessed by the completion time of `TUW_Scatterv` (or `MPI_Scatterv`) being smaller than the guideline time.

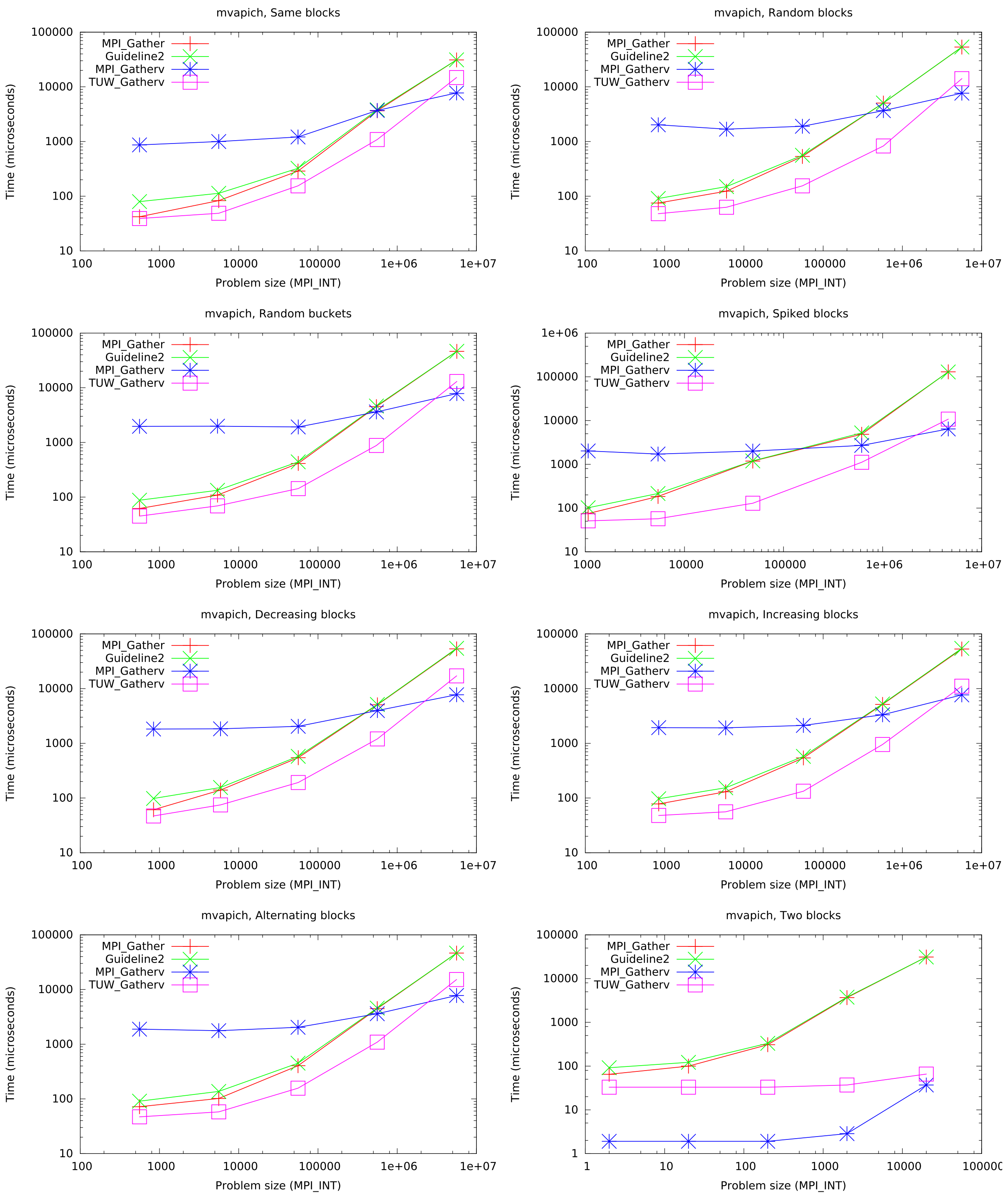

Fig. 5. Results for MVAPICH with $p = 35 \times 16 = 560$ processes for (padded) `MPI_Gather`, right-hand side of Guideline (GL2), `MPI_Gatherv` and `TUW_Gatherv`. Minimum completion times over 75 repetitions are in microseconds (μs), problem sizes in number of `MPI_INT` elements. Fulfillment of Guideline (GL2) is witnessed by the completion time of `TUW_Gatherv` (or `MPI_Gatherv`) being smaller than the guideline time.

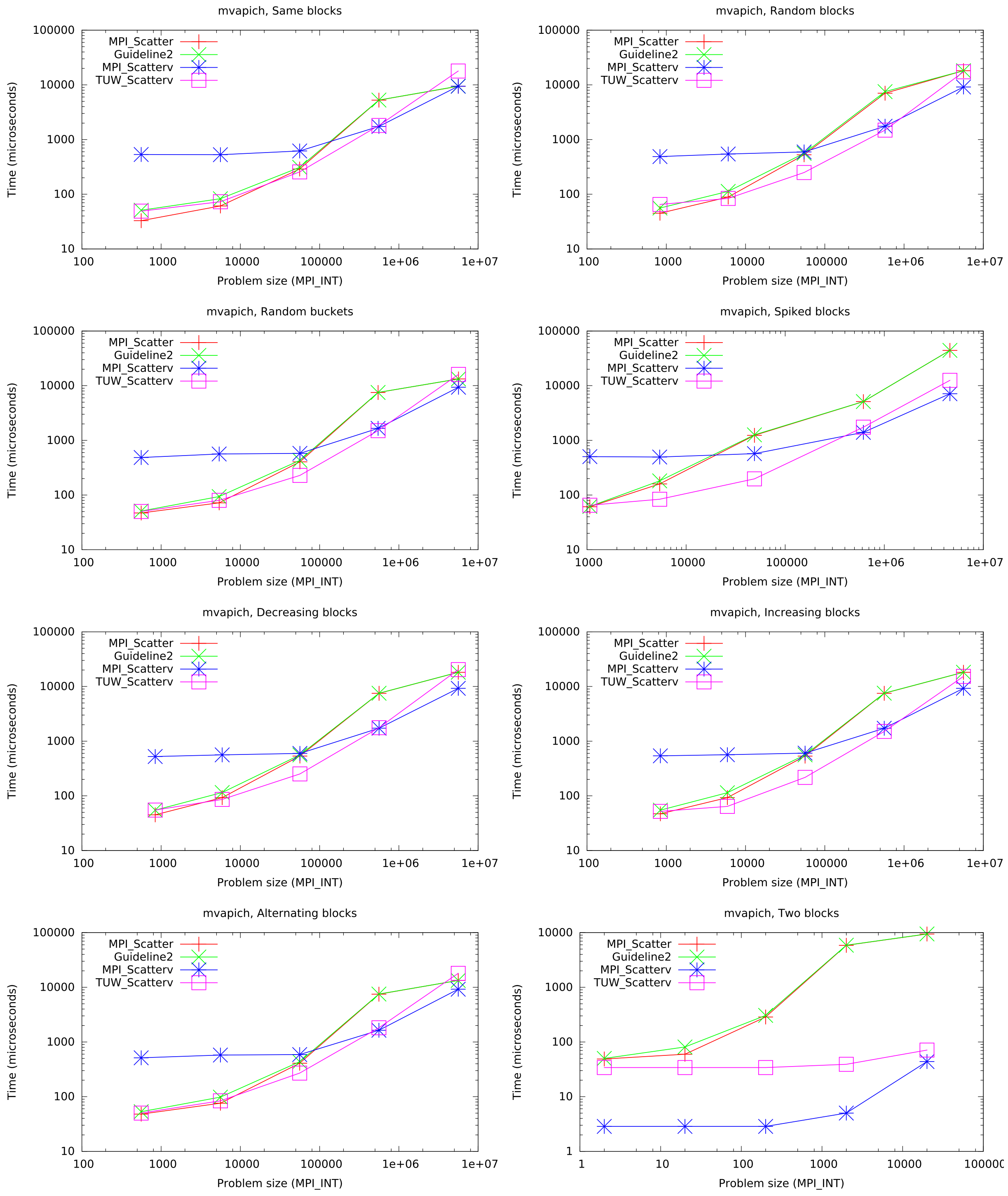

Fig. 6. Results for MVAPICH with $p = 35 \times 16 = 560$ processes for (padded) `MPI_Scatter`, right-hand side of Guideline (GL2), `MPI_Scatterv` and `TUW_Scatterv`. Minimum completion times over 75 repetitions are in microseconds (μs), problem sizes in number of `MPI_INT` elements. Fulfillment of Guideline (GL2) is witnessed by the completion time of `TUW_Scatterv` (or `MPI_Scatterv`) being smaller than the guideline time.

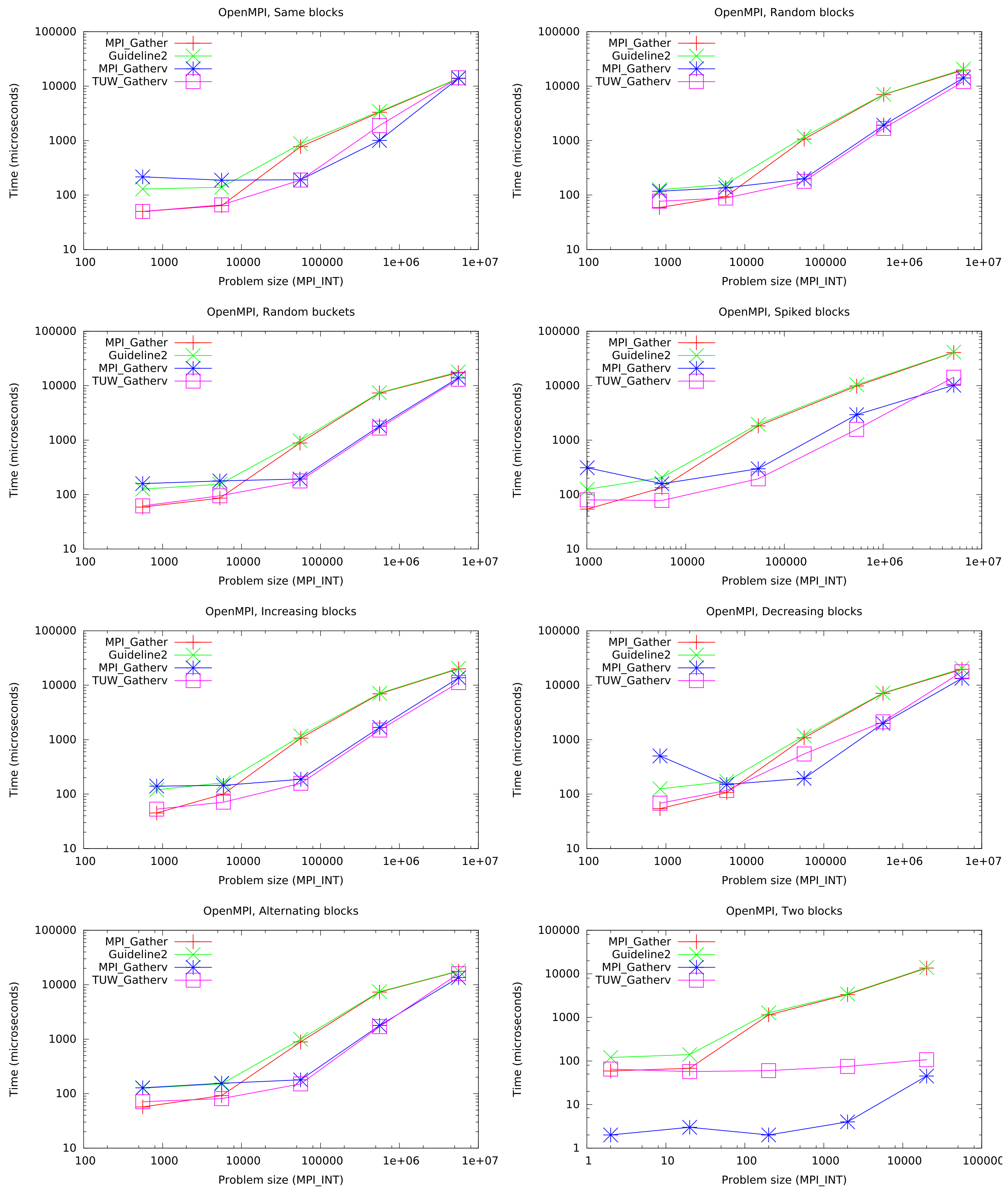

Fig. 7. Results for Open MPI with $p = 35 \times 16 = 560$ processes for (padded) `MPI_Gather`, right-hand side of Guideline (GL2), `MPI_Gatherv` and `TUW_Gatherv`. Minimum completion times over 75 repetitions are in microseconds (μs), problem sizes in number of `MPI_INT` elements. Fulfillment of Guideline (GL2) is witnessed by the completion time of `TUW_Gatherv` (or `MPI_Gatherv`) being smaller than the guideline time.

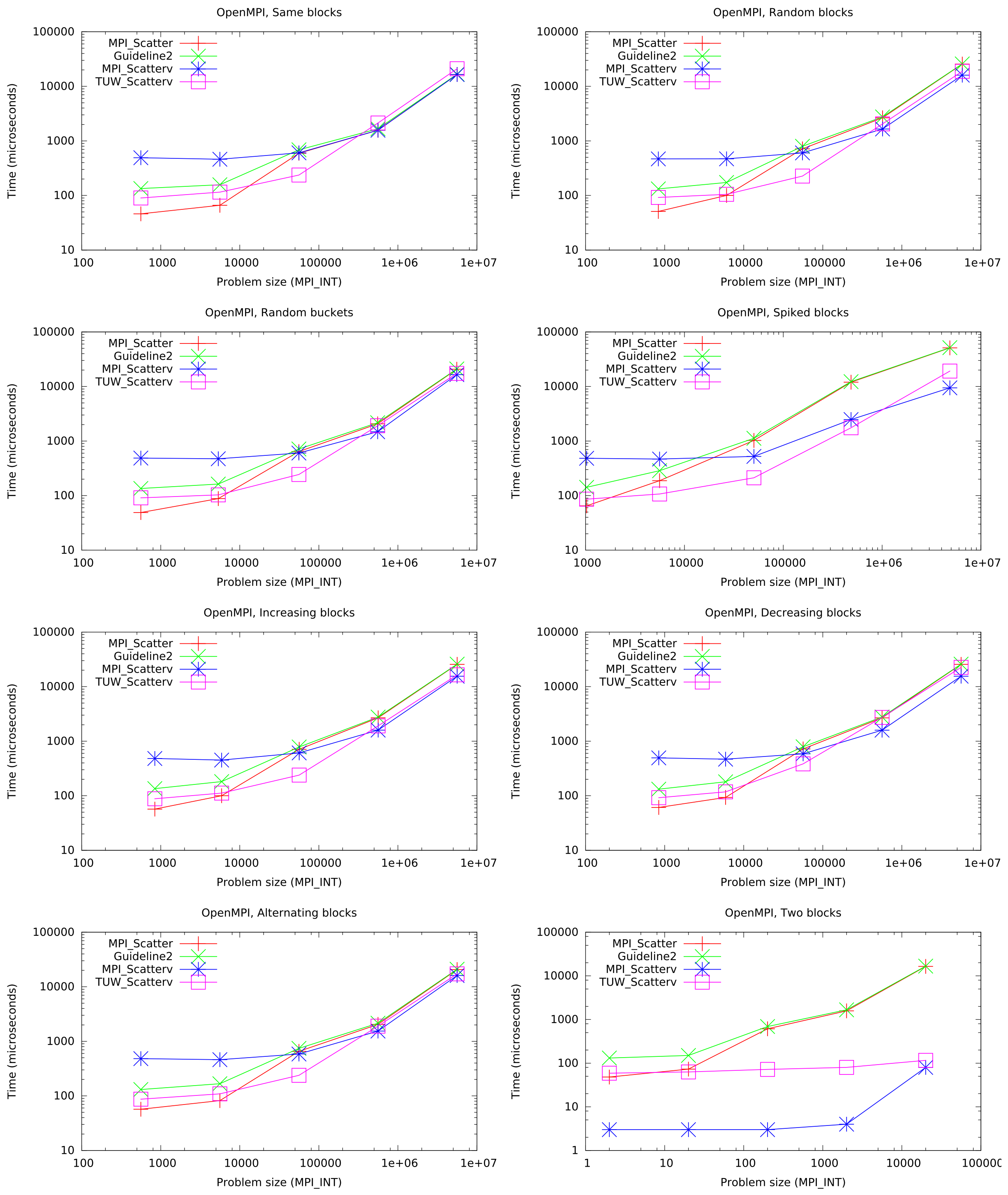

Fig. 8. Results for Open MPI with $p = 35 \times 16 = 560$ processes for (padded) `MPI_Scatter`, right-hand side of Guideline (GL2), `MPI_Scatterv` and `TUW_Scatterv`. Minimum completion times over 75 repetitions are in microseconds (μs), problem sizes in number of `MPI_INT` elements. Fulfillment of Guideline (GL2) is witnessed by the completion time of `TUW_Scatterv` (or `MPI_Scatterv`) being smaller than the guideline time.

For the smaller problem instances, all libraries fail Guideline (GL2) with their MPI_Gatherv implementations, and, with the exception of the NEC MPI library, for their MPI_Scatterv implementations as well, and by very large factors. Only for the **two blocks** problems, the linear algorithms behind MPI_Gatherv and MPI_Scatterv can meet the performance guideline. The TUW_Gatherv implementation fulfill Guideline (GL2) by a large margin, as does TUW_Scatterv except for the largest instance for the **same** problem.

Our second system is a medium-large InfiniBand/Intel cluster consisting of 2000 Dual Intel Xeon E5-2650v2 8-core processors running at 2.6 GHz, interconnected with an InfiniBand QDR-80 network.[3] The MPI library is Intel MPI 2017.1 and the compiler is Intel MPI 2017.1 with optimization level -O3. The benchmarks were executed first with the default environment for this machine, which pins the MPI processes to the 16 cores per node, and makes choices for which MPI_Gather and MPI_Gatherv implementations to use. This led to excessively poor results, as documented in the technical report accompanying this paper [24], rendering our TUW_Gatherv implementation several orders of magnitude faster for small problems than the native MPI_Gather and MPI_Gatherv, badly violating the trivial Guideline (GL1). Much better performance could be achieved by choosing other implementations offered by the Intel MPI 2017.1 library; good choices for the experiments here seems to be a binomial tree algorithm for MPI_Gather, a k-nomial tree for MPI_Gatherv, a binomial tree for MPI_Scatter and a linear algorithm for MPI_Scatterv (no k-nomial tree implementation available). These choices were effected by setting the appropriate I_MPI_ADJUST_ environment variables. The results with these implementation choices are shown in Figs. 9 and 10. Experiments were run with $p = 500 \times 16 = 8000$ MPI processes, and we plot the minimum completion times (in microseconds) over 75 repetitions.

We first observe that TUW_Gatherv and TUW_Scatterv have similar completion times over the problem types, except that the **increasing** problem type is slower to complete than the other problem types. The gather results show that while a binomial tree algorithm can work well also for MPI_Gatherv for almost regular problems, there is a penalty as the problems get more irregular, which the TUW_Gatherv implementation pays to a much lesser extent. The TUW_Gatherv implementation in all cases outperforms MPI_Gatherv for the Intel MPI 2017.1 library by a factor of two to three, noteworthy especially for the **two blocks** problem. The TUW_Gatherv implementation shows no violations of Guideline (GL2), which the native MPI_Gatherv implementation does for most of the problems for some, mostly small problem sizes, and for the **same** problem for the whole range. As can also be seen from the **same** problem, the Intel MPI 2017.1 library violates Guideline (GL1) with TUW_Gatherv being faster than MPI_Gather, indicating an inferior implementation of the binomial tree algorithm. The linear algorithm used for MPI_Scatterv leads to severe violations of Guideline (GL2) for the smaller problems sizes, which TUW_Scatterv easily fulfills, except for the mid-sized **same** problems. The TUW_Scatterv implementation is faster than MPI_Scatterv by huge factors for all smaller problems, except for the **two blocks** problem where the linear algorithm with no tree construction overhead has a slight advantage. For the large problem sizes from $b = 1000$, the linear MPI_Scatterv is faster than TUW_Scatterv. This indicates that it would make sense to extend the algorithms for the irregular problems proposed here to gracefully degrade towards linear behavior as block sizes increase [5]. For the IntelMPI library, the overhead for the tree construction in TUW_Gatherv and TUW_Scatterv as given by the two blocks problem for $b = 1$ is about 30 μs.

6. Conclusion

This paper described new, simple algorithms for performing irregular gather and scatter operations as found in MPI in linear communication time and logarithmic number of communication start-up's, a considerable improvement over both fixed, data oblivious logarithmic depth trees and direct communication with the root. An experimental evaluation on two InfiniBand-based systems shows that the resulting implementations can, especially for overall small problem instances be considerably faster than current MPI library MPI_Gatherv and MPI_Scatterv implementations by large factors, and provide guideline consistent performance over a range of differently structured, irregular problems. Our prototype implementations can readily be incorporated into existing MPI libraries. The algorithms were derived under the assumption of homogeneous, linear communication costs, which has often been pointed out to be inadequate for, e.g., hierarchically structured systems [25]. It would easily be possible to use the new algorithms in a hierarchical fashion and still have overall linear-time performance; indeed, it would be interesting and worthwhile to implement and benchmark good, hierarchical implementations of the whole set of MPI gather and scatter operations.

Although the trees constructed by our algorithm are provably good, they are structured as binomial trees, and thus incur $\lceil \log_2 p \rceil$ communication operations at the root. In an asynchronous communication model, it is sometimes possible to do slightly better, and have fewer than logarithmic communication start-ups at the root. In [14] it is shown that optimal, ordered gather and scatter trees can still be computed (offline) in polynomial time. However, these algorithms are not practical, and the possible improvements over the algorithms in this paper are entirely marginal.

The tree construction technique of Lemma 3 can be applied to other problems as well, for instance to construct good, problem dependent trees for sparse reduction operations [26].

[3] This is the so-called Vienna Scientific Cluster, the VSC3, see vsc.ac.at. The author thanks for support and access to this machine.

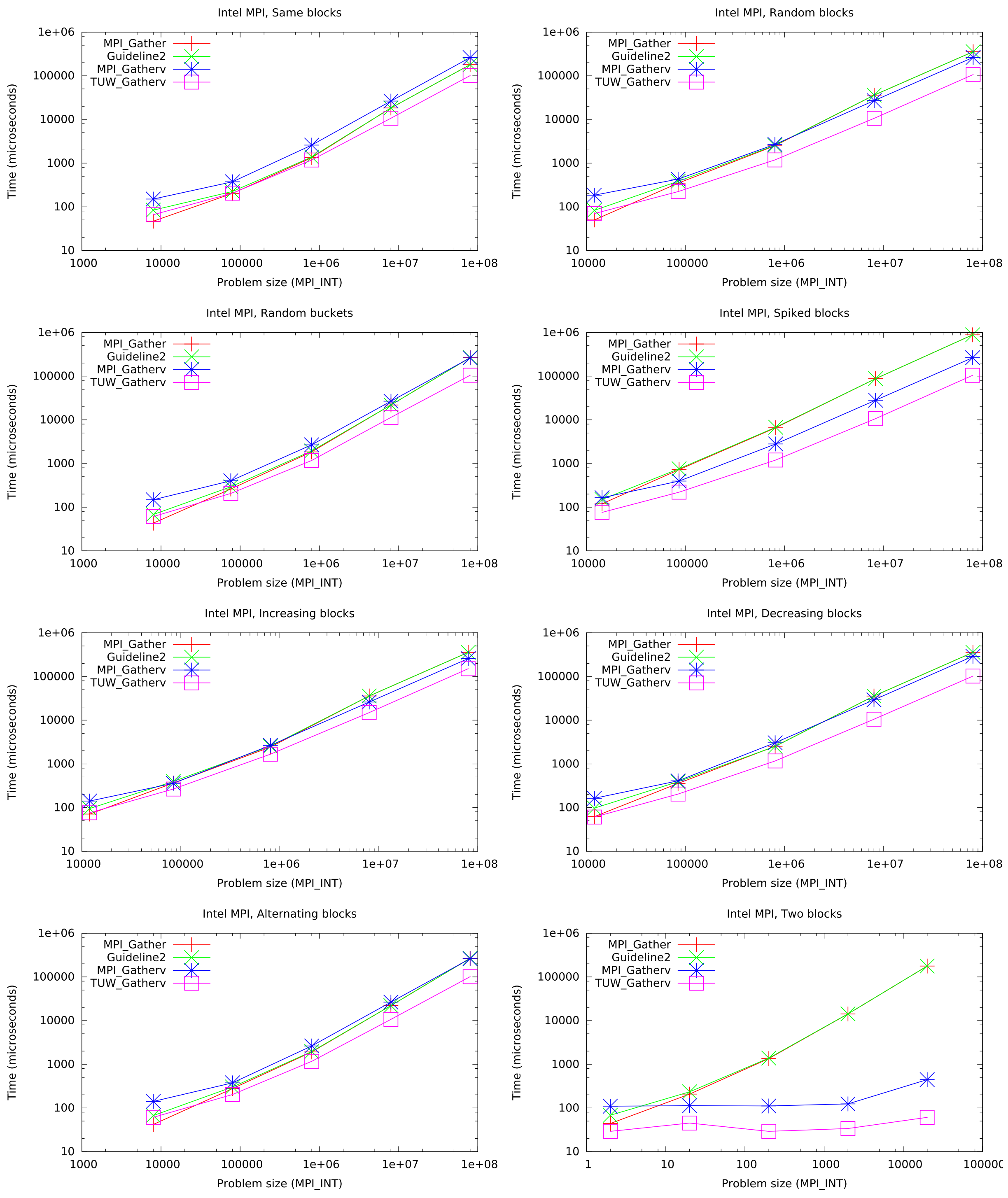

Fig. 9. Results for Intel MPI with $p = 500 \times 16 = 8000$ processes for (padded) `MPI_Gather`, right-hand side of Guideline (GL2), `MPI_Gatherv` and `TUW_Gatherv`. Minimum completion times over 75 repetitions are in microseconds (µs), problem sizes in number of `MPI_INT` elements. Fulfillment of Guideline (GL2) is witnessed by the completion time of `TUW_Gatherv` (or `MPI_Gatherv`) being smaller than the guideline time.

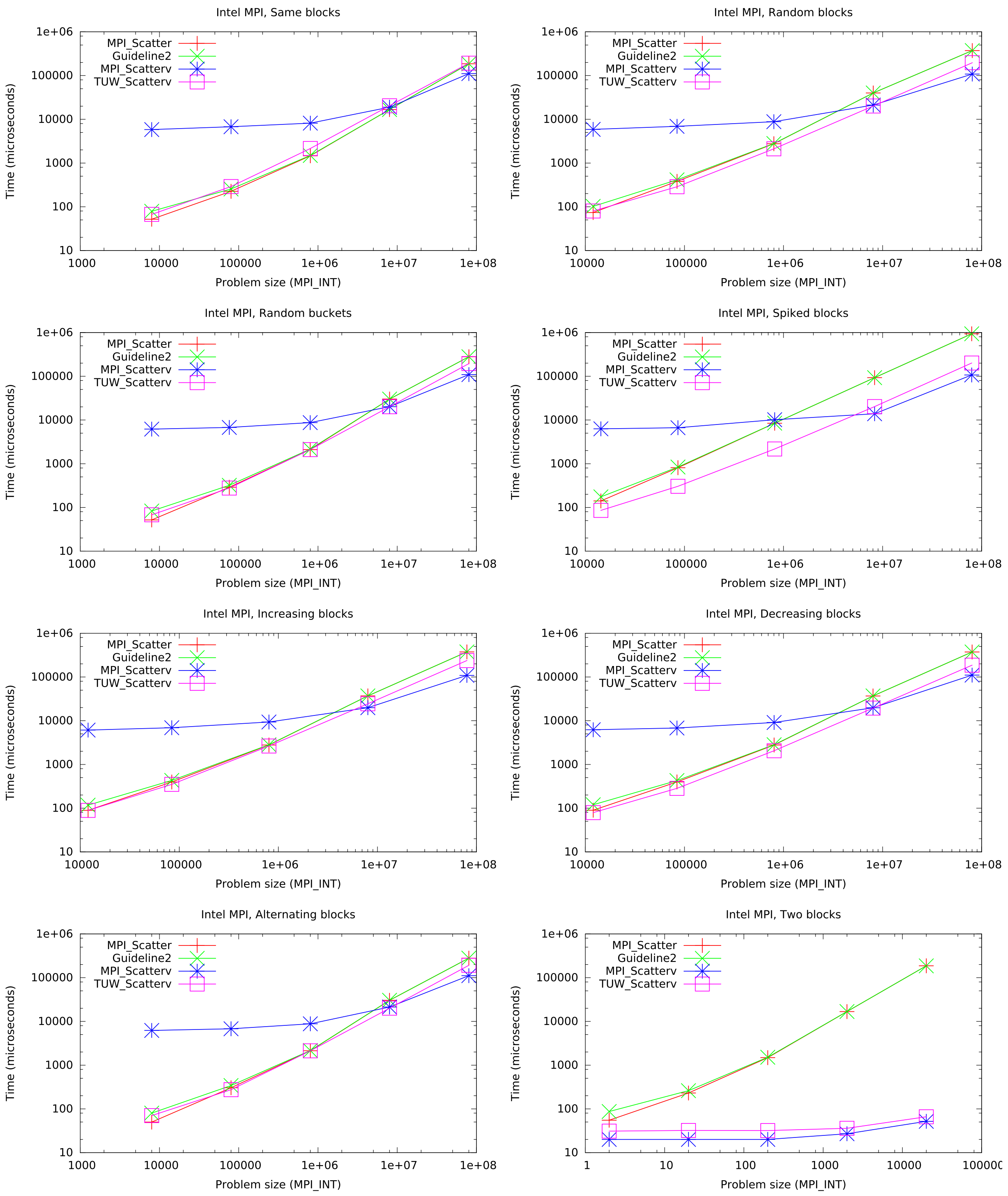

Fig. 10. Results for Intel MPI with $p = 500 \times 16 = 8000$ processes for (padded) `MPI_Scatter`, right-hand side of Guideline (GL2), `MPI_Scatterv` and `TUW_Scatterv`. Minimum completion times over 75 repetitions are in microseconds (μs), problem sizes in number of `MPI_INT` elements. Fulfillment of Guideline (GL2) is witnessed by the completion time of `TUW_Scatterv` (or `MPI_Scatterv`) being smaller than the guideline time.

References

[1] J.L. Träff, Practical, linear-time, fully distributed algorithms for irregular gather and scatter, in: 23rd European MPI Users' Group Meeting (EuroMPI/USA), ACM, 2017, pp. 1–10.
[2] MPI Forum, MPI: A message-passing interface standard. version 3.1, 2015, www.mpi-forum.org.
[3] E. Chan, M. Heimlich, A. Purkayastha, R.A. van de Geijn, Collective communication: theory, practice, and experience, Concurr. Comput. 19 (13) (2007) 1749–1783.
[4] Y. Saad, M.H. Schultz, Data communication in parallel architectures, Parallel Comput. 11 (2) (1989) 131–150.
[5] J.L. Träff, Hierarchical gather/scatter algorithms with graceful degradation, in: 18th International Parallel and Distributed Processing Symposium (IPDPS), 2004, p. 80.
[6] K. Dichev, V. Rychkov, A.L. Lastovetsky, Two algorithms of irregular scatter/gather operations for heterogeneous platforms, in: Recent Advances in the Message Passing Interface; 17th European MPI Users' Group Meeting (EuroMPI), Springer, 2010, pp. 289–293. Volume 6305 of Lecture Notes in Computer Science.
[7] Z. Ben-Miled, J.A.B. Fortes, R. Eigenmann, V.E. Taylor, On the implementation of broadcast, scatter and gather in a heterogeneous architecture, in: Thirty-First Annual Hawaii International Conference on System Sciences (HICSS), 1998, pp. 216–225.
[8] J. Hatta, S. Shibusawa, Scheduling algorithms for efficient gather operations in distributed heterogeneous systems, in: Proceedings of the 2000 International Workshop on Parallel Processing (ICPPW), IEEE, 2000, pp. 173–180.
[9] L. Boxer, R. Miller, Coarse grained gather and scatter operations with applications, J. Parallel Distrib. Comput. 64 (11) (2004) 1297–1310.
[10] H.P. Charles, P. Fraigniaud, Scheduling a scattering-gathering sequence on hypercubes, Parallel Process. Lett. 3 (1993) 29–42.
[11] S. Shibusawa, H. Makino, S. Nimiya, J. Hatta, Scatter and gather operations on an asynchronous communication model, in: Proceedings of the 2000 ACM Symposium on Applied Computing (SAC), ACM, 2000, pp. 685–691.
[12] S.N. Bhatt, G. Pucci, A. Ranade, A.L. Rosenberg, Scattering and gathering messages in networks of processors, IEEE Trans. Comput. 42 (8) (1993) 938–949.
[13] J. Bruck, C.T. Ho, S. Kipnis, E. Upfal, D. Weathersby, Efficient algorithms for all-to-all communications in multiport message-passing systems, IEEE Trans. Parallel Distrib. Syst. 8 (11) (1997) 1143–1156.
[14] J.L. Träff, On optimal trees for irregular gather and scatter collectives, 2017. arXiv:1711.08731.
[15] S. Hunold, A. Carpen-Amarie, F.D. Lübbe, J.L. Träff, Automatic verification of self-consistent MPI performance guidelines, in: Euro-Par Parallel Processing, Vol. 9833 of Lecture Notes in Computer Science, 2016, pp. 433–446.
[16] J.L. Träff, W.D. Gropp, R. Thakur, Self-consistent MPI performance guidelines, IEEE Trans. Parallel Distrib. Syst. 21 (5) (2010) 698–709.
[17] J.L. Träff, A library for advanced datatype programming, in: 23rd European MPI Users' Group Meeting (EuroMPI), ACM, 2016, pp. 98–107.
[18] T. Hoefler, S. Gottlieb, Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using MPI datatypes, in: Recent Advances in Message Passing Interface. 17th European MPI Users' Group Meeting, Vol. 6305 of Lecture Notes in Computer Science, Springer, 2010, pp. 132–141.
[19] J.L. Träff, A. Rougier, S. Hunold, Implementing a classic: Zero-copy all-to-all communication with MPI datatypes, in: 28th ACM International Conference on Supercomputing (ICS), ACM, 2014, pp. 135–144.
[20] S. Hunold, A. Carpen-Amarie, Tuning MPI collectives by verifying performance guidelines, in: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia), 2018, pp. 64–74.
[21] A. Carpen-Amarie, S. Hunold, J.L. Träff, On expected and observed communication performance with MPI derived datatypes, Parallel Comput. 69 (2017) 98–117.
[22] R. Thakur, W.D. Gropp, R. Rabenseifner, Improving the performance of collective operations in MPICH, Int. J. High Perform. Comput. Appl. 19 (2005) 49–66.
[23] W. Gropp, E. Lusk, Reproducible measurements of MPI performance characteristics, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting, Vol. 1697 of Lecture Notes in Computer Science, Springer, 1999, pp. 11–18.
[24] J.L. Träff, Practical, linear-time, fully distributed algorithms for irregular gather and scatter, 2017. arXiv:1702.05967.
[25] W. Gropp, L.N. Olson, P. Samfass, Modeling MPI communication performance on SMP nodes: Is it time to retire the ping pong test, in: 23rd European MPI Users' Group Meeting (EuroMPI), ACM, 2016, pp. 41–50.
[26] J.L. Träff, Transparent neutral element elimination in MPI reduction operations, in: Recent Advances in Message Passing Interface. 17th European MPI Users' Group Meeting, Vol. 6305 of Lecture Notes in Computer Science, Springer, 2010, pp. 275–284.