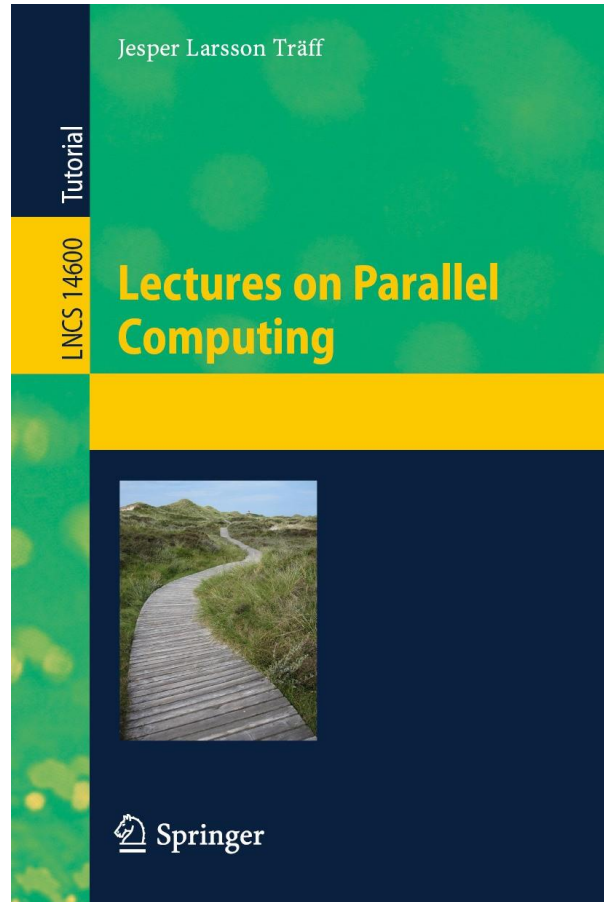


Effective MPI: User-defined Datatypes and Cartesian Communicators for All-to-all Communication in Multidimensional Tori*

Jesper Larsson Träff
traff@par.tuwien.ac.at

TU Wien (Vienna University of Technology)
Faculty of Informatics, Institute of Computer Engineering
Research Group Parallel Computing

*Dedicated to Bill Gropp, on occasion of the UIUC Bill Gropp Symposium



With substantial chapters on

- OpenMP
- MPI

Free copies available (a few)...

Springer LNCS 14600, 2026
(although complete in 2024)

What is this about...

MPI has powerful, orthogonal features that seem somehow related:

Collectives, derived datatypes, communicators, attributes, and persistence... (not mentioned in paper)

We give a concrete (new) multidimensional torus-all-to-all algorithm that uses all these features together in a meaningful way

Outcome:

Generic, tunable/mapable zero-copy MPI_Alltoall algorithm and implementation, useful in many contexts

Three (or 4) viewpoints:

1. A dedicated algorithm for d-dimensional, 1-ported tori
2. An p-processor algorithm for any factorization of p into d factors
3. (A scheme for hierarchical algorithms and implementations for the MPI collectives. **Not this talk**)
4. A performance guideline (implementation) for MPI_Alltoall: MPI_Alltoall in terms of MPI_Alltoall on smaller sets of processes (and other MPI functionality)

Ad 4: Experimental results show problems with/features of common MPI_Alltoall implementations

Ad 2: Generalization of standard, message-combining hypercube all-to-all algorithm

Ad 3: Generalizes multi-lane collective algorithms and implementations

Jesper Larsson Träff, Sascha Hunold: Decomposing MPI Collectives for Exploiting Multi-lane Communication. CLUSTER 2020: 270-280

The all-to-all algorithm

Factor $p = p_0 \times p_1 \times \dots \times p_{d-1}$ into d factors

Let $D[i] = p_i$ to be used as array dimensions

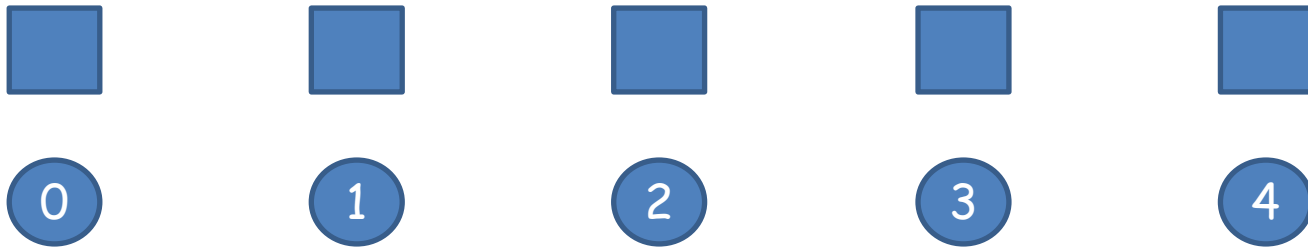
Procedure AlltoAll($S[p], R[p], G$)

$G=(V,E)$ communication structure (graph, e.g., d -dimensional torus), $|V|=p$

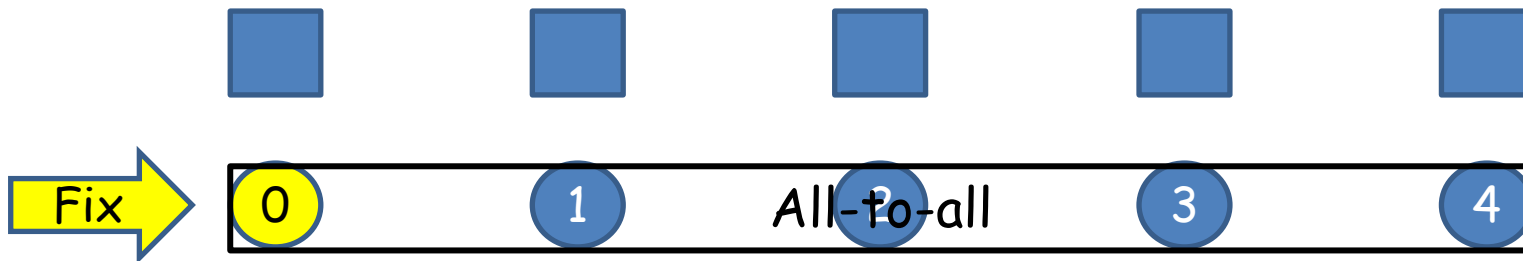
$S[p]$: Array of p sendblocks, symmetrically, $S[i]$ is the block to processor i

$R[p]$: Array of p receiveblocks, $R[i]$ is the block from processor i

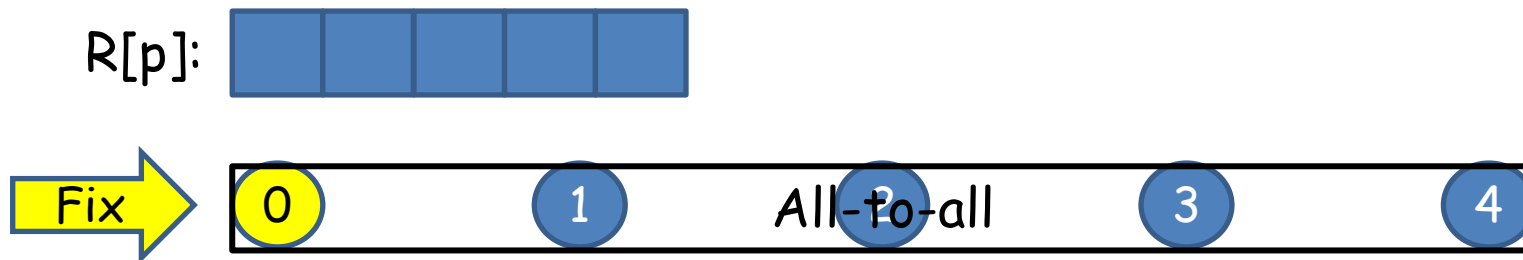
All-to-all in $d=1$ dimension (example: $p=5$)



All-to-all in $d=1$ dimension (example: $p=5$)



All-to-all in d=1 dimension (example: p=5)



Blockindices, order of blocks

$R[p] = [0,1,2,3,4]$

$S[p] = [0,1,2,3,4]$

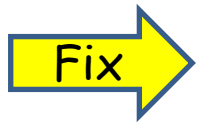
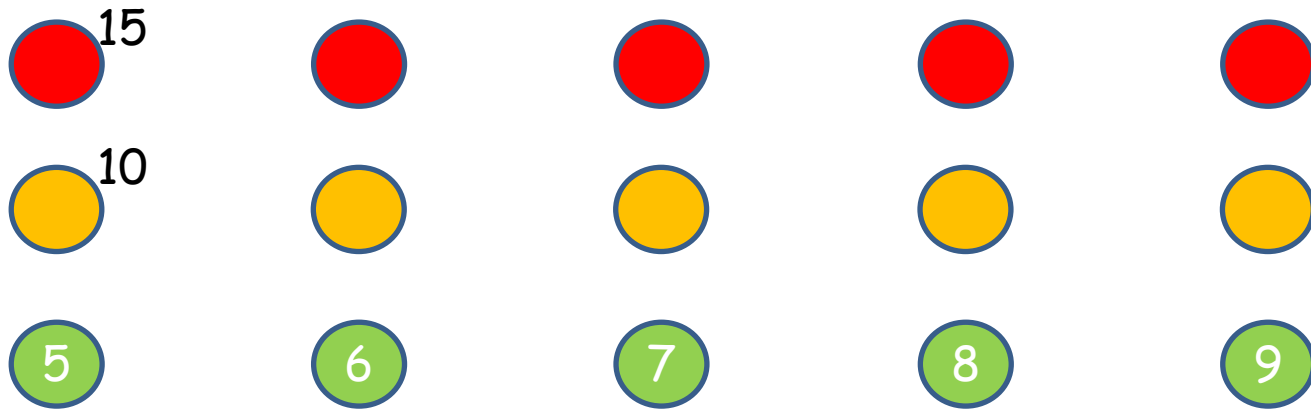
Indices and traversal order
used in communication, e.g., all-
to-all

All-to-all in $d=2$ dimensions (example: $p=5 \times 4=20$)

- All-to-all along dimension 0, $D[0]=5$, $4=D[1]$ simultaneous all-to-all operations (row-wise)
- All-to-all along dimension 1, $D[1]=4$, $5=D[0]$ simultaneous all-to-all operations (column-wise)

Keep in mind: All processors communicate symmetrically

All-to-all in d=2 dimensions (example: $p=5 \times 4=20$)

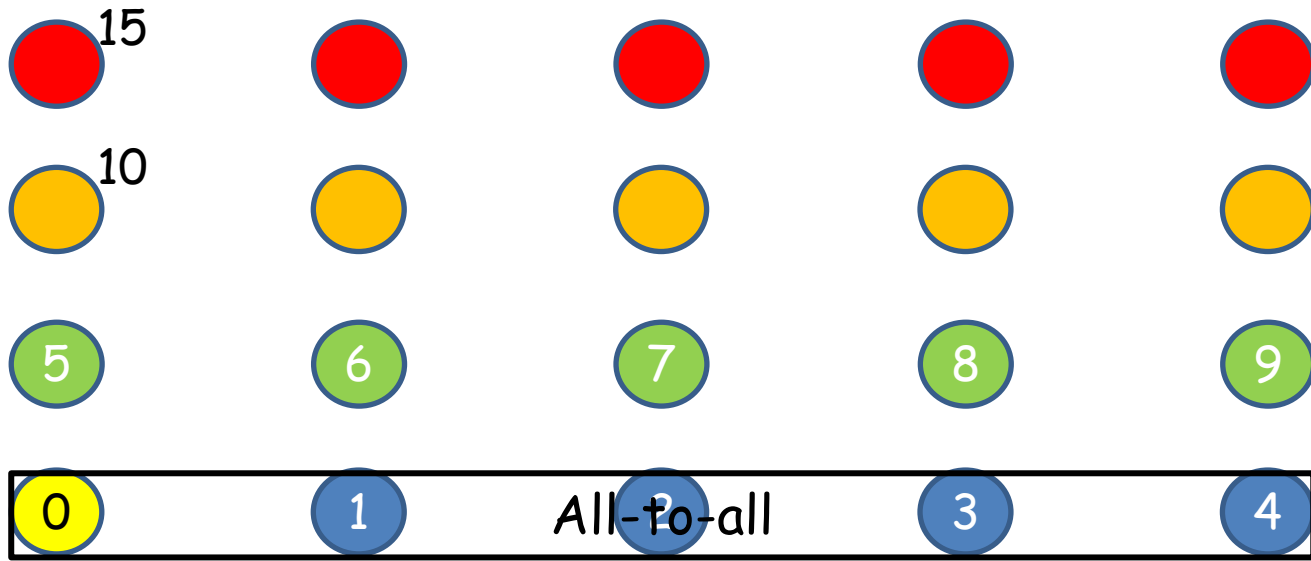


R[p]:



From processor 0

All-to-all in d=2 dimensions (example: $p=5 \times 4=20$)

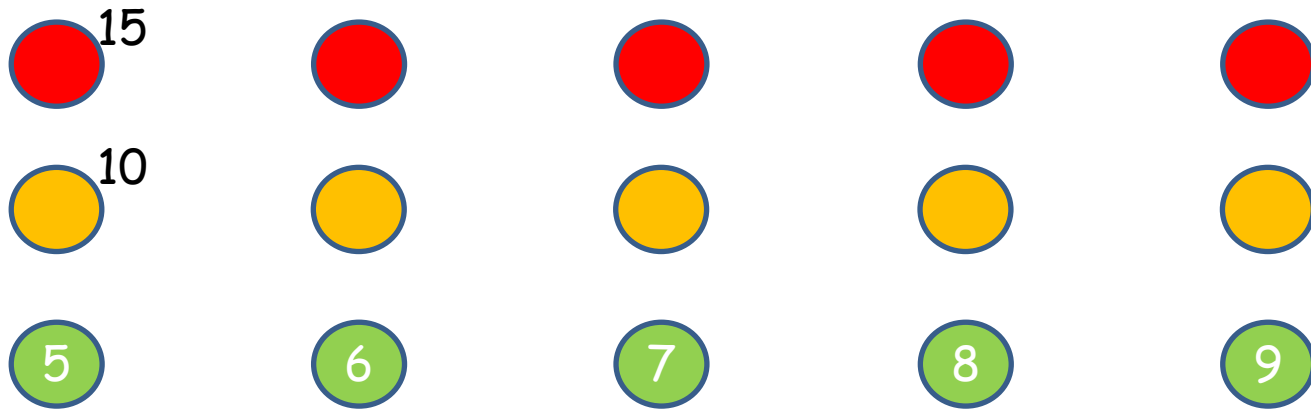


$R[p]:$



From processor 0, 1

All-to-all in d=2 dimensions (example: $p=5 \times 4=20$)

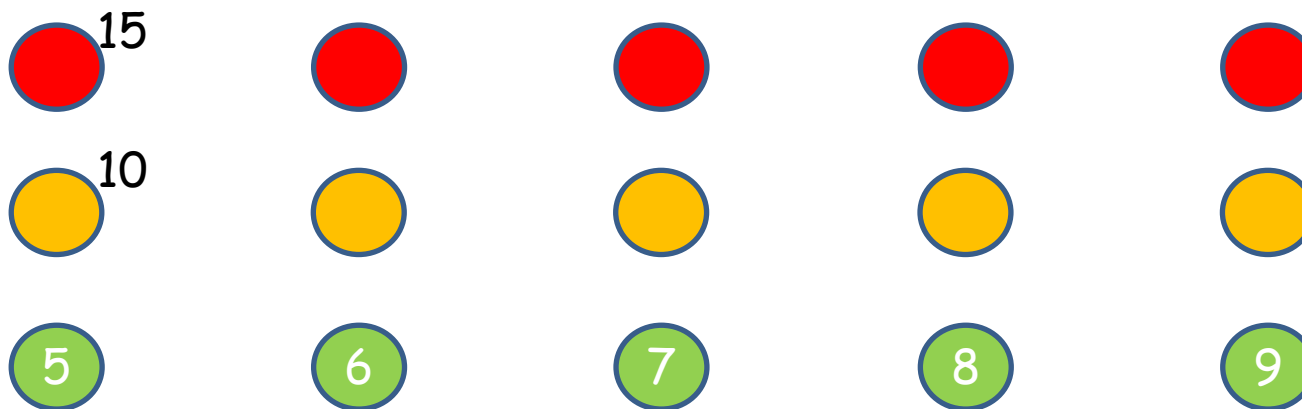


$R[p]:$



From processor 0, 1, 2

All-to-all in d=2 dimensions (example: $p=5 \times 4=20$)

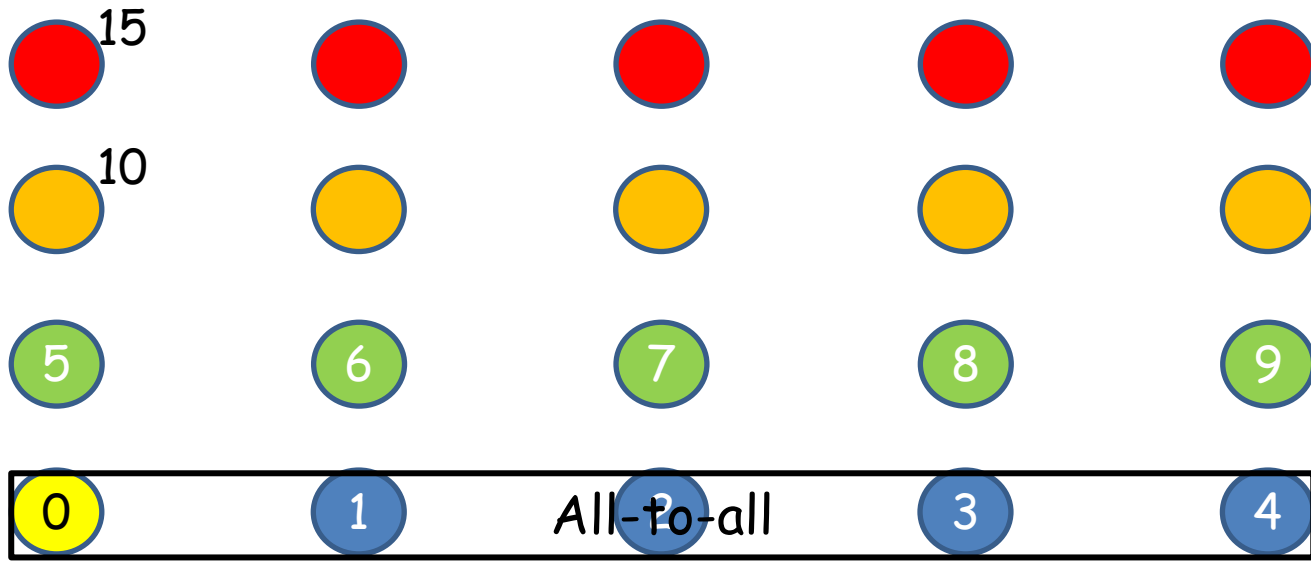


$R[p]:$



From processor 0, 1, 2, 3

All-to-all in d=2 dimensions (example: $p=5 \times 4=20$)

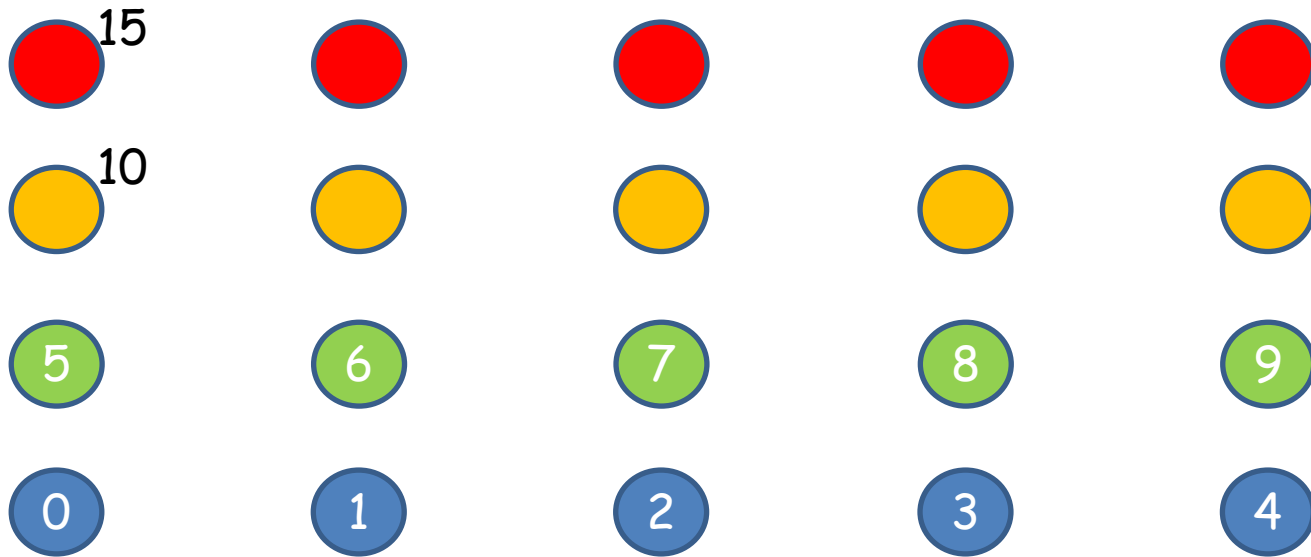


$R[p]:$



From processor 0, 1, 2, 3, 4

All-to-all in d=2 dimensions (example: $p=5 \times 4=20$)

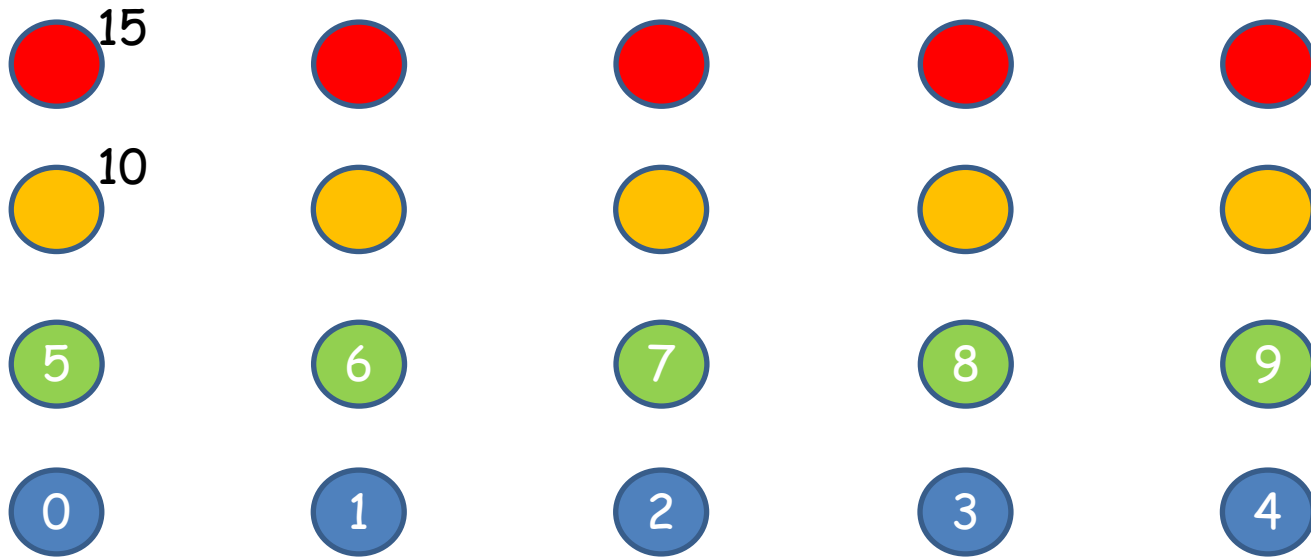


R[p]:



Stride: $D[0]=5$ blocks

All-to-all in d=2 dimensions (example: $p=5 \times 4=20$)

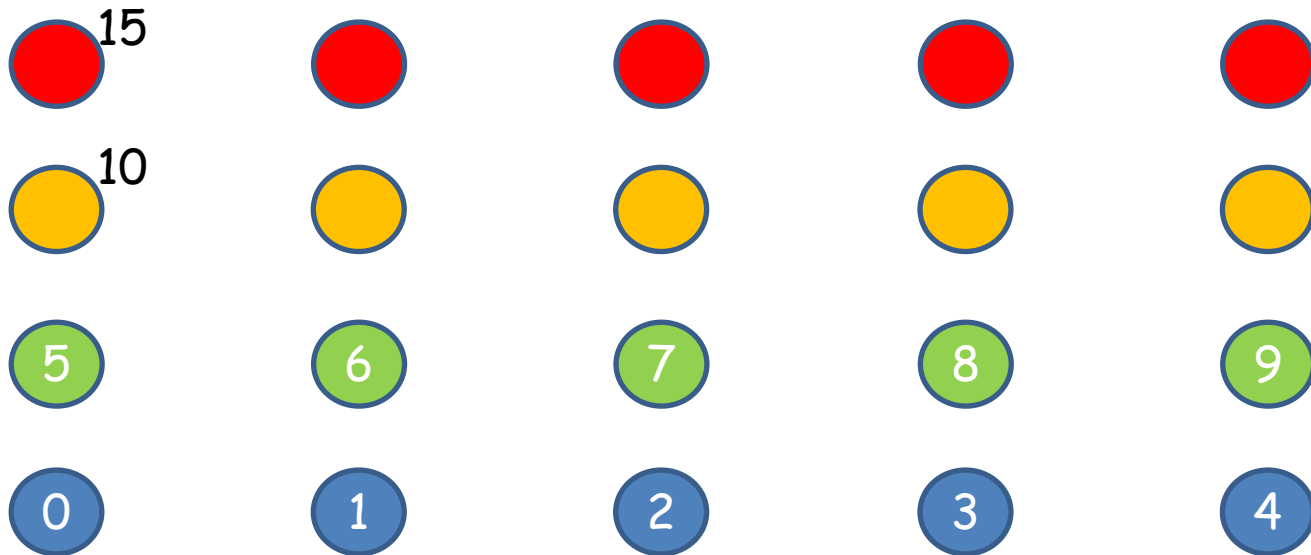


Blockindices, order of blocks for dimension 0 all-to-all
 $R[p] == R'[D[0]]_1[D[1]]_5 = [0,1,2,3,4][0,5,10,15]$



Stride for this dimension

All-to-all in d=2 dimensions (example: $p=5 \times 4=20$)



$R'[0] = [0, 5, 10, 15]$

$R'[1] = [1, 6, 11, 16]$

$R'[2] = [2, 7, 12, 17]$

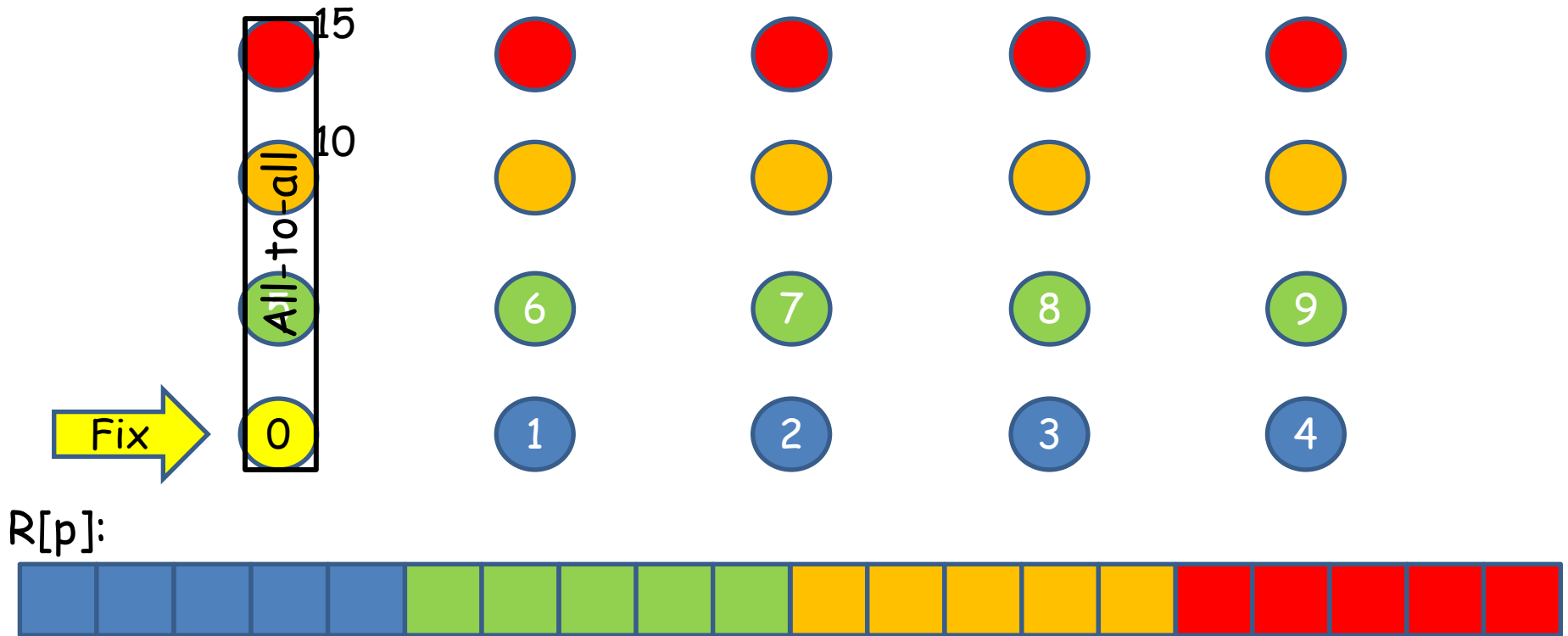
$R'[3] = [3, 8, 13, 18]$

$R'[4] = [4, 9, 14, 19]$

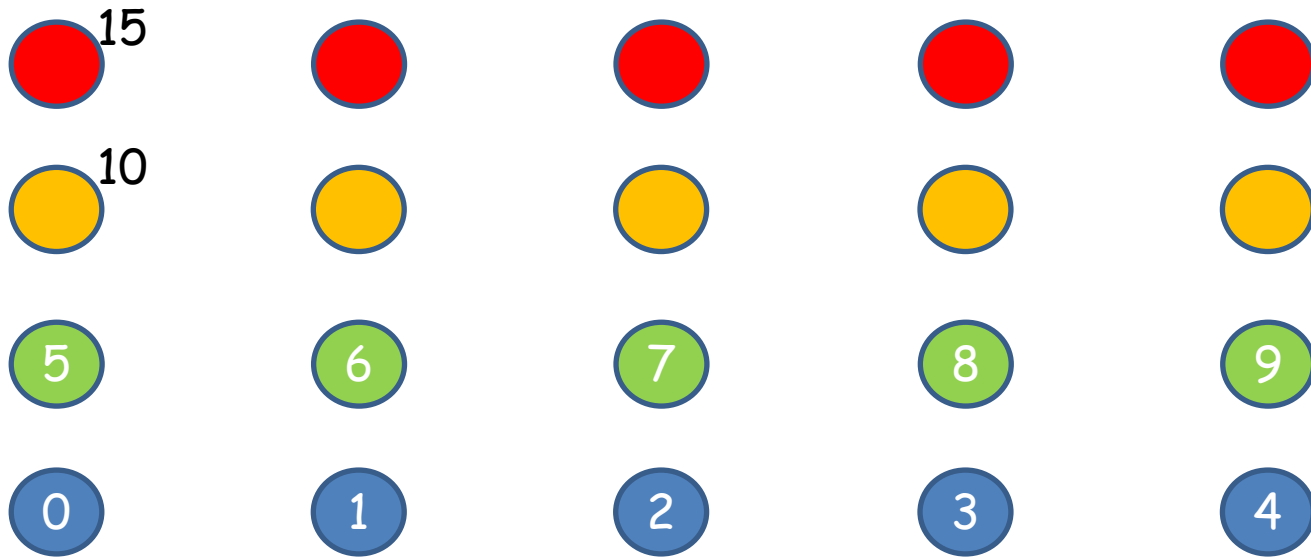
$S'[0] = \dots$ exactly the same

Use two buffers for intermediate blocks, $T[0] == S'$, $T[1] = R'$, and alternate

All-to-all in d=2 dimensions (example: $p=5 \times 4=20$)



All-to-all in d=2 dimensions (example: $p=5 \times 4=20$)



Blockindices, order of blocks for dimension 1 all-to-all
 $R[p] == R'[D[1]]_5 = [0,5,10,15]$



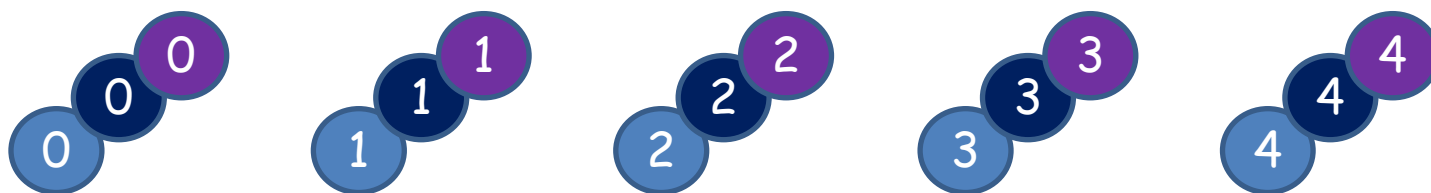
Stride for this dimension, blocksize is 5
 $= p/D[1]$

All-to-all in $d=3$ dimensions (example: $p=5 \times 4 \times 3=60$)

- All-to-all along dimension 0, $D[0]=5$, $4 \times 3 = D[1] \times D[2]$
simultaneous all-to-all operations
- All-to-all along dimension 1, $D[1]=4$, $5 \times 3 = D[0] \times D[2]$
simultaneous all-to-all operations
- All-to-all along dimension 2, $D[2]=3$, $5 \times 4 = D[0] \times D[1]$
simultaneous all-to-all operations

Keep in mind: All processors communicate symmetrically

All-to-all in d=3 dimensions (example: $p=5 \times 4 \times 3=60$)



Blockindices, order of blocks for dimension 0 all-to-all

$R[p] == R'[D[0]]_1[D[1]]_5[D[2]]_{20} = [0,1,2,3,4][0,5,10,15][0,20,40]$,
blocksize 1

Blockindices, order of blocks for dimension 1 all-to-all

$R[p] == R'[D[1]]_5[D[2]]_{20} = [0,5,10,15][0,20,40]$, blocksize 5

Blockindices, order of blocks for dimension 2 all-to-all

$R[p] == R'[D[2]]_{20} = [0,20,40]$, blocksize 20

Notation:

Multi-dimensional array R' aliased with one dimensional array R ,
For each dimension d and each index i , $0 \leq i < d$

$$R'[d_0]_{s_0}[d_1]_{s_1}[d_2]_{s_2} \dots [i_0][i_1][i_2] \dots = R[i_0 s_0 + i_1 s_1 + i_2 s_2 + \dots]$$

Blockindices, order of blocks for dimension 0 all-to-all

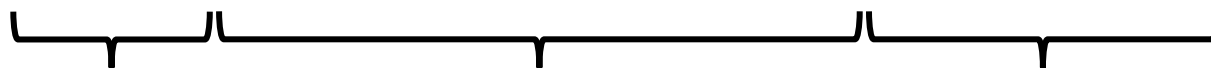
$$R[p] == R'[D[0]]_1[D[1]]_5[D[2]]_{20} = [0,1,2,3,4][0,5,10,15][0,20,40]$$

$\underbrace{\hspace{1.5cm}}$ $\underbrace{\hspace{3.5cm}}$

No. of blocks for dimension 0 Structure of block, size is $D[1]D[2] = p/D[0]$

The general algorithm: Each processor does

- d rounds of all-to-all communication along dimension k , $0 \leq k < d$
- In round k , alias $R[p]$ with

$$R'[D[k]]_{\sigma(k)}[D[k+1]]_{\sigma(k+1)} \dots [D[d-1]]_{\sigma(d-1)}$$


where $\sigma(k) = \prod_{0 \leq i < k} D[i]$; same for $S[p]$

- All-to-all along dimension k containing the processor, subgraph of Cartesian product graph $G = (V_0, E_0) \square (V_1, E_1) \square \dots \square (V_{d-1}, E_{d-1})$

Cartesian product graph $G = (V_0, E_0) \square (V_1, E_1) \square \dots \square (V_{d-1}, E_{d-1})$, G_k subgraph for each processor for dimension k

```

Procedure AlltoAll( $S[p], R[p], G$ )
 $O = \text{RankVector}(\text{rank}, d, \text{comm})$ 
 $B = 0$ 
 $T[b] = S$  // copy into temporary buffer (can be avoided)
for ( $k=0, 1, \dots, d-1$ ) {
     $S'[D[k]]_{\sigma(k)}[D[k+1]]_{\sigma(k+1)} \dots [D[d-1]]_{\sigma(d-1)} == T[b]$ 
     $R'[D[k]]_{\sigma(k)}[D[k+1]]_{\sigma(k+1)} \dots [D[d-1]]_{\sigma(d-1)} == T[1-b]$ 
     $G_k = \{O[0]\} \times \{O[1]\} \times \dots \times \{O[k-1]\} \times V_k \times \{O[k+1]\} \times \dots \times \{O[d-1]\}$ 
    AlltoAll( $S', R', G_k$ )
     $b = 1-b$ 
}
 $R = T[b]$  // copy out of temporary buffer (can be avoided)

```

$$\sigma(k) = \prod_{0 \leq i < k} D[i]$$

Theorem:

Given a factorization of p into d factors, and a factorization of the communication structure G into d factors, the all-to-all problem can be solved in d communication rounds. In round k , $D[k]-1$ blocks of size $(p/D[k])$ per processor are communicated, in total $\sum p(D[k]-1)/D[k]$. This is best possible.

Invariant for correctness:

Before round k , each processor with coordinates $(o_0, o_1, \dots, o_{d-1})$ has collected a block to each of the processors

$$V_{d-1} \times \dots \times V_k \times \{o_{k-1}\} \times \dots \times \{o_0\}$$

from each of the processors in

$$\{o_{d-1}\} \times \dots \times \{o_k\} \times V_{k-1} \times \dots \times V_0$$

Observations:

Zero-copy algorithm: No explicit processor local data movement, only array aliasing and change of traversal order

In last round, dimension $d-1$, the composite blocks are consecutive (see 2-dimension example)

Special cases:

- For $d=\log_2 p$, p power of two, the algorithm is the standard, message combining hypercube algorithm, optimal volume $dp/2$ blocks per processor. Small factors generalize hypercube
- For $d=2$, the algorithm is the two-level hierarchical large-algorithm

Jesper Larsson Träff, Sascha Hunold: Decomposing MPI Collectives for Exploiting Multi-lane Communication. CLUSTER 2020: 270-280

MPI at work

1. Factoring p (size in comm) into d factors
2. Creating dimension-wise subcommunicators
3. Caching for reuse
4. d MPI_Alltoall communication rounds:
5. Create datatypes for strided access
6. Caching for reuse?

Persistent comm

Persistent arguments?

1. Factorizing p into d factors

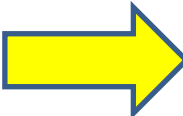
Either factorization is already given (system constraint), or

```
int order[d];  
MPI_Dims_create(p,d,order);
```

Computes a factorization of p into d factors, subject to certain closeness constraints; may or may not be useful...

Jesper Larsson Träff, Felix Donatus Lübke: Specification
Guideline Violations by `MPI_Dims_create`. EuroMPI 2015: 19:1-
19:2

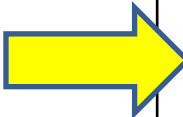
2. Creating subcommunicators



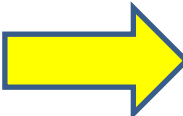
```
int periodicity [d];
for (i=0; i<d; i++) periodicity [i] = 1;
// make Cartesian communicator with no reorder
MPI_Cart_create (comm,d,order,periodicity,0,factorcomm);

factors = (torusattr*)malloc(sizeof(torusattr));
assert (factors != NULL);
factors->d = d;
factors->comm = (MPI_Comm*)malloc(d*sizeof(MPI_Comm));

int subdim[d];
for (k=0; k<d; k++) subdim[k] = 0;
for (k=0; k<d; k++) {
    subdim[k] = 1;
    MPI_Cart_sub(*factorcomm,subdim,&factors->comm[k]);
    subdim[k] = 0;
}
```



3. Caching for reuse (amortization)



```
MPI_Comm_set_attr (*factorcomm, toruskey (), factors );

static int toruskey () {
    static int toruskeyval = MPI_KEYVAL_INVALID ; // hidden key
    if (toruskeyval == MPI_KEYVAL_INVALID) {
        MPI_Comm_create_keyval (MPI_COMM_NULL_COPY_FN ,
                                torusdel, &toruskeyval , NULL);
    }
    return toruskeyval ;
}
```



4. d all-to-all communication rounds

```
int Alltoall_torus (void *sendbuf,  
                  int sendcount, MPI_Datatype sendtype,  
                  void *recvbuf,  
                  int recvcount, MPI_Datatype recvtype,  
                  MPI_Comm comm)  
{  
    int size; // process does not need to know its rank  
  
    ...  
  
    MPI_Aint lb, blockextent ;  
    MPI_Type_contiguous(recvcount,recvtype,& block);  
    MPI_Type_get_extent(block,&lb,&blockextent); count = 1;
```



```
void * tempbuf ;  
tempbuf = (void*)malloc(size*blockextent);  
assert(tempbuf != NULL );  
int flag ;  
torusattr * factors ;  
MPI_Comm_get_attr (comm,toruskey(),&factors,&flag );
```

look

```
for (i=factors->d -1; i >=0; i --) {  
    stride [i] = count ;  
    MPI_Comm_size(factors->comm [i],&c);  
    count = c* count ;  
}
```

Precompute array strides

```
void *in , *out ;  
out = sendbuf ;  
in = (factors ->d%2==0) ? tempbuf : recvbuf ;
```

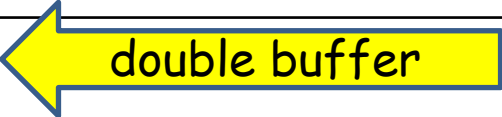
double buffer



```
count = 1;
for (k=factors->d -1; k >=0; k --) {
    MPI_Datatype segment, tile, tiled;
    MPI_Type_contiguous(count,block,&segment);
    for (i=0; i<k; i++) { // column - major parts
        MPI_Type_create_resized(segment,0,
                                stride[i]*blockextent,&tile);

        MPI_Type_free(&segment);
        MPI_Comm_size(factors->comm[i],&c);
        MPI_Type_contiguous(c,tile,&segment);
        MPI_Type_free(&tile);
    }
    MPI_Type_create_resized(segment,0,count*blockextent,
                            &tiled);

    MPI_Type_free(&segment);
    MPI_Type_commit(&tiled);
    MPI_Alltoall (out,1,tiled,in,1,tiled,factors->comm[k]);
    MPI_Type_free (&tiled);
}
```



```
if (out==sendbuf) {  
    if (in==recvbuf) {  
        out = recvbuf; in = tempbuf ;  
    } else {  
        out = tempbuf; in = recvbuf ;  
    }  
} else {  
    void *s;  
    s = in; in = out; out = s;  
}  
MPI_Comm_size(factors->comm[k],&c);  
count = c*count;  
}  
MPI_Type_free(&block);  
free (tempbuf);  
return MPI_SUCCESS ;  
}
```

Comments:

- True **zero-copy implementation**: No local copies, double buffering with sendbuf as first input and recvbuf as last result
- Only contiguous/resize datatype constructors!
- Symmetric: In most of the code, a process doesn't even need to know its rank
- Created datatypes **cannot be cached**: dependent on input count/datatype
- **Except for persistent Alltoall_init... might be worth a try**
- **Tuning options**: Factorization/mapping, choice of MPI_Alltoall implementation

Some experimental results

Small 36x32 core cluster, dual socket Intel(R) Xeon(R) Gold 6130F 16-core CPUs at 2.10GHz, dual Intel Omnipath 100Gbit/s.

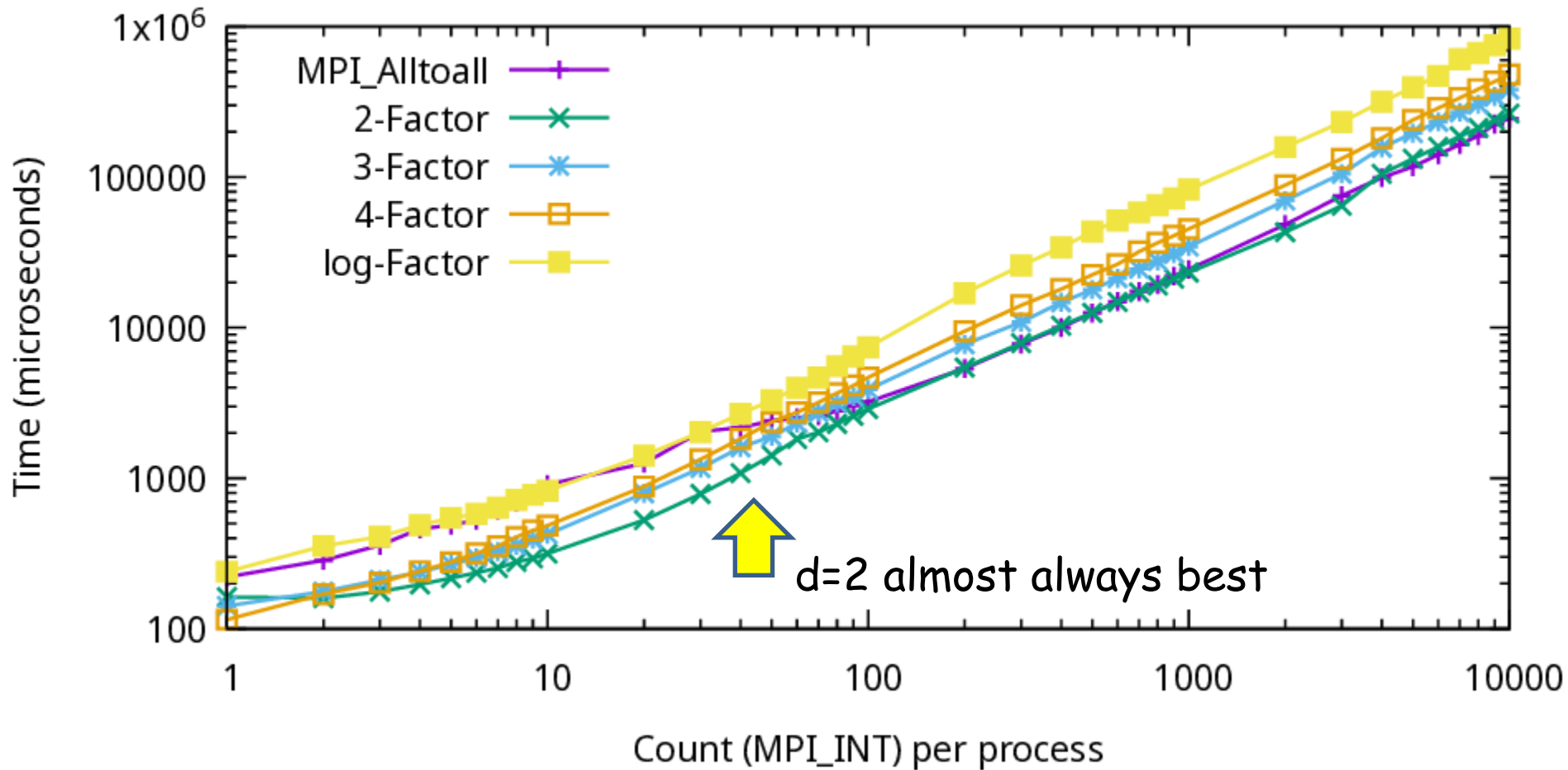
Code compiled with gcc 10.2.1 with the -O3 option.

The MPI libraries used are OpenMPI 4.1.4, mpich 4.1.2 (with gcc 12.1.0), and IntelMPI 2021.13.

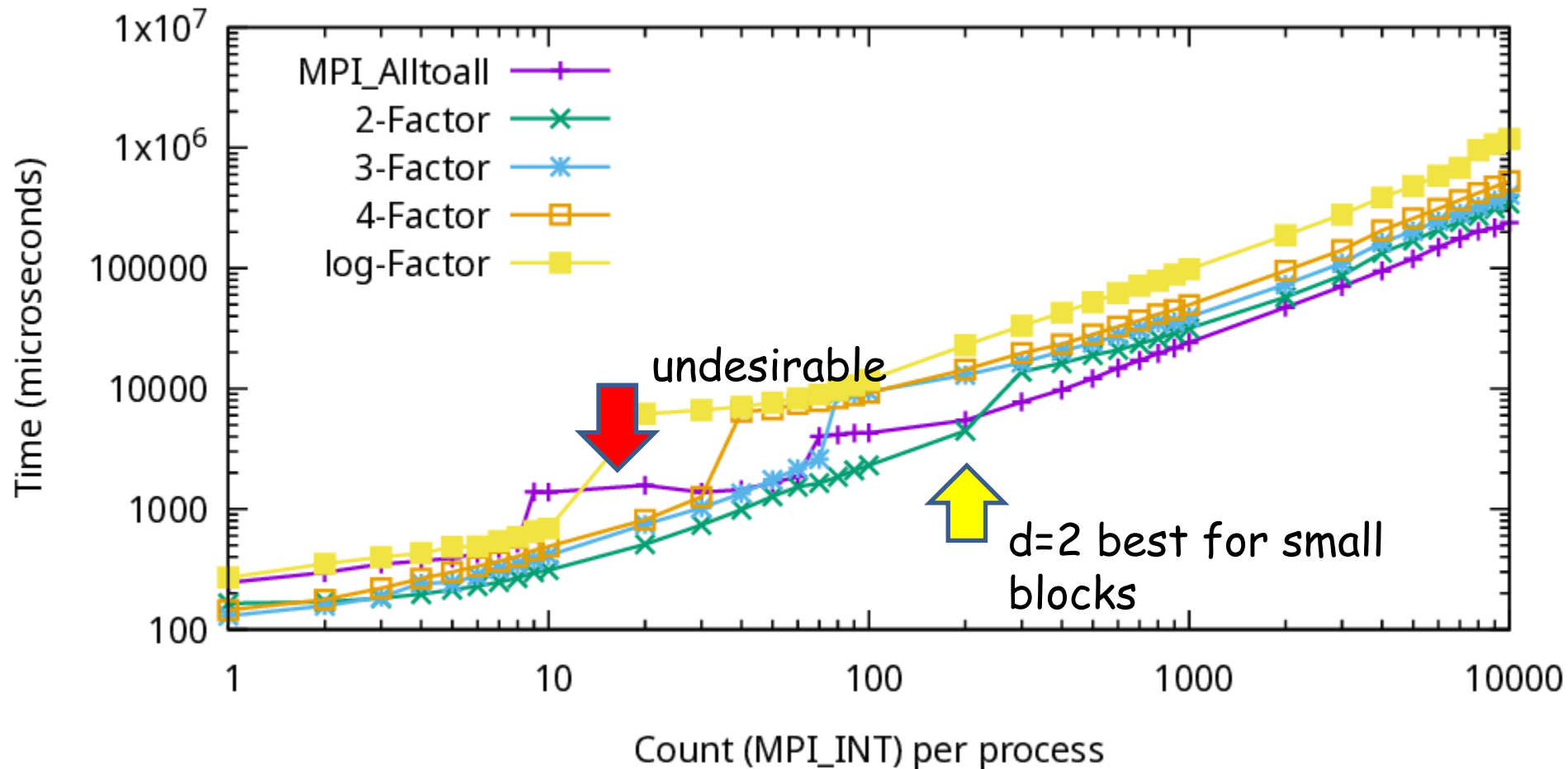
Experiments with full 36x32 core system, 1152 MPI processes, MPI_INT, count=1,...,10.000, 40 repetitions (+8 warmups) per measurement, synchronization with double MPI_Barrier

Different factorizations $d=2,3,4,\text{ceil}(\log_2 p)$ against native MPI_Alltoall

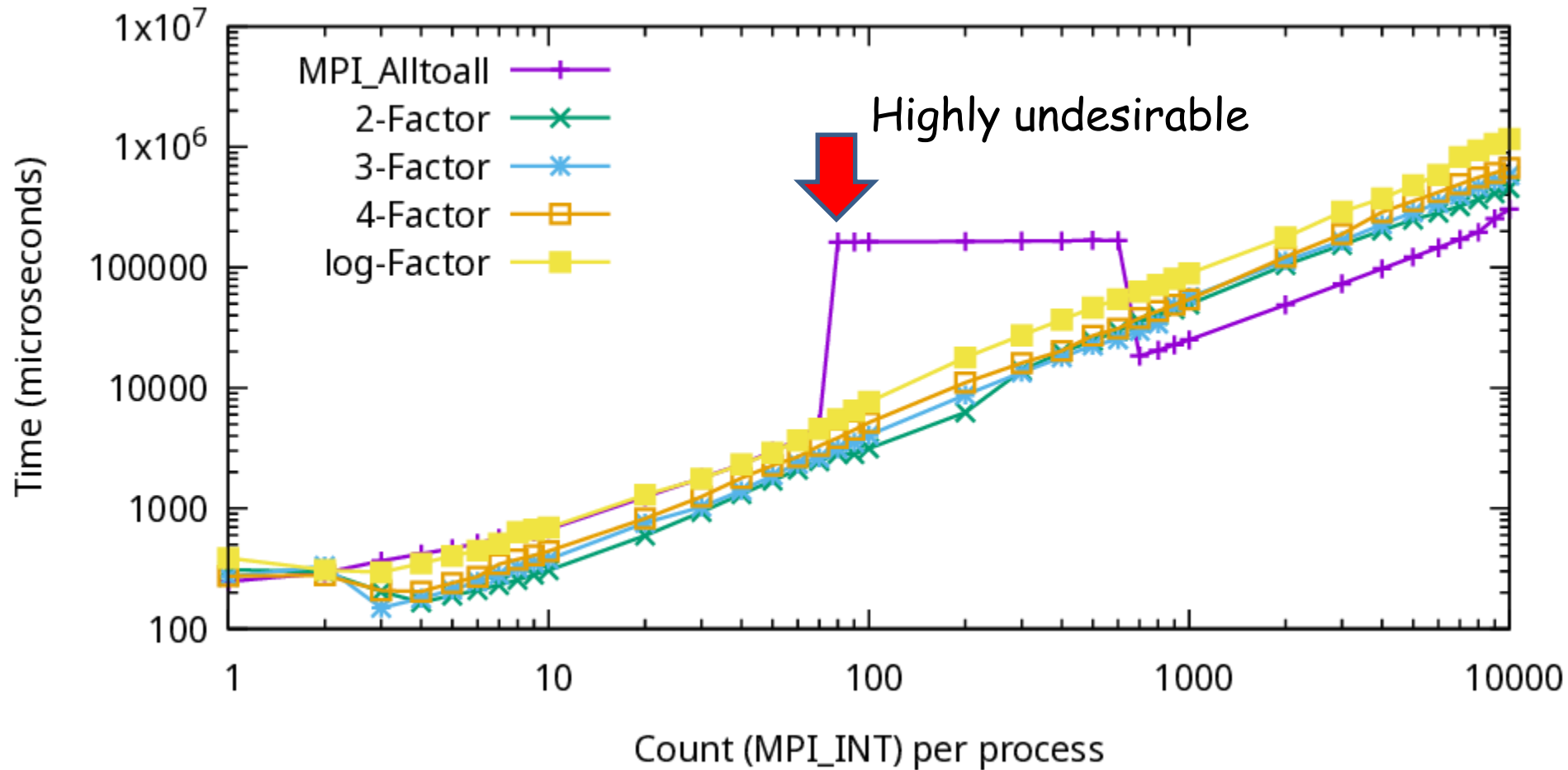
MPI_Alltoall, 36x32 MPI processes, mpich library



MPI_Alltoall, 36x32 MPI processes, Intel MPI library



MPI_Alltoall, 36x32 MPI processes, OpenMPI library



Summary

- Textbook implementation of somewhat complex algorithm by using the rich MPI features: datatypes, communicators
- Interesting tuning possibilities, indicates room for improvement in common libraries
- Good datatype performance essential for efficient zero-copy algorithms

Much more on this and many other things in

Jesper Larsson Träff:
Algorithms for Collective Communication
(deterministic parallel computing on distributed memory machines)
MIT Press, 2027