Workshop for Communication Architecture in Clusters, IPDPS 2002:

Exploiting Transparent Remote Memory Access for Non-Contiguous- and One-Sided-Communication

Joachim Worringen, Andreas Gäer, Frank Reker

RWTH Scalable Computing Aachen

Lehrstuhl für Betriebssysteme, RWTH Aachen

Agenda

- Introduction into SCI via PCI
- MPI via SCI: SCI-MPICH
- Non-Contiguous Datatype Communication
- One-sided Communication
- Conclusion

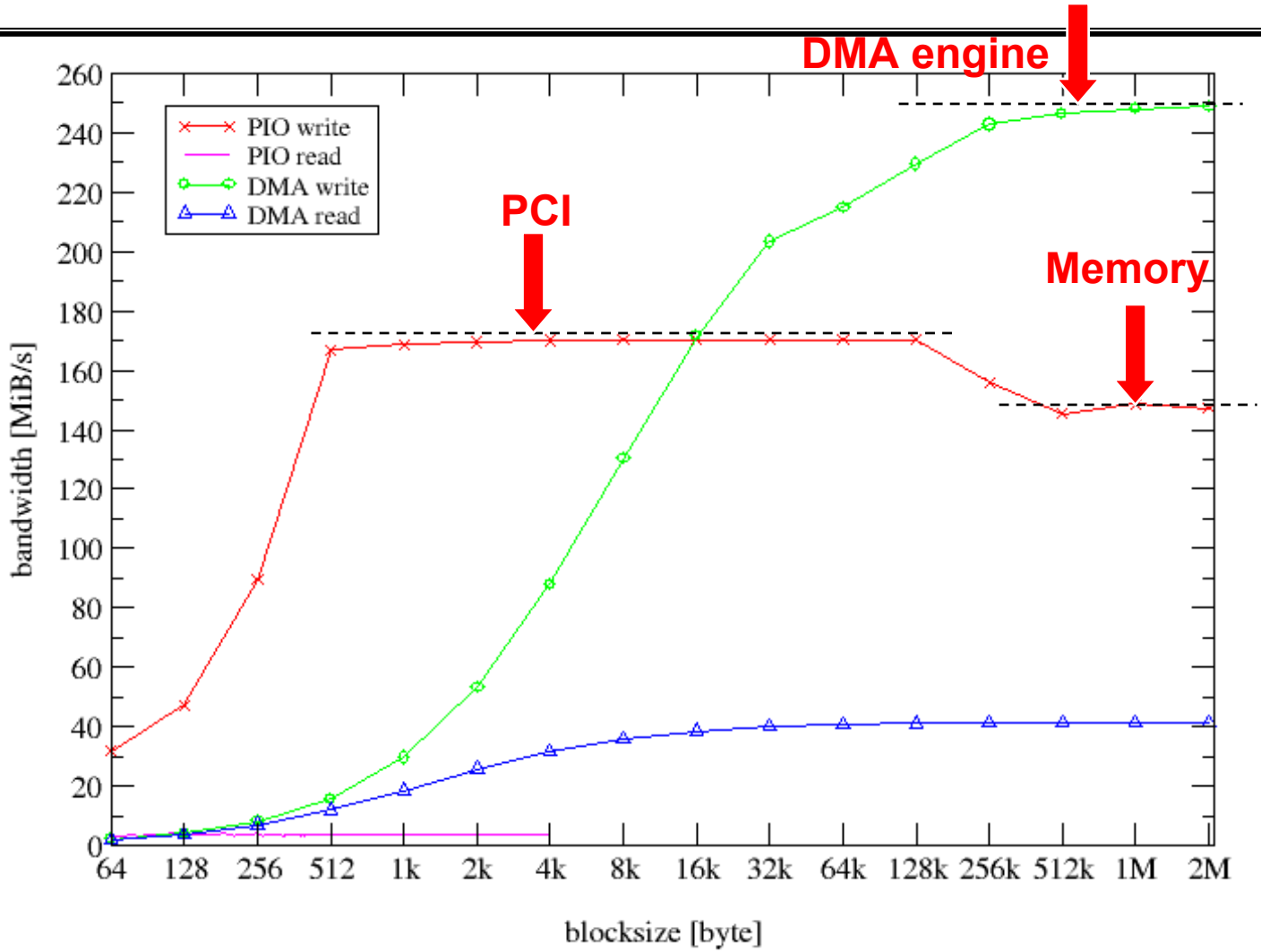
SCI – Principle

- **S**calable **C**oherent **I**nterface:
  Protocol for a **memory-coupling interconnect**
- standardized in 1992 (IEEE 1596-1992)
- **Not** a physical bus – but offers bus-like services
- transparent operation for source and target
- operates within a global **64-bit adress space**
- Good scalability through flexible topologies and small packet sizes
- **optional:** efficient, distributed cache-coherence

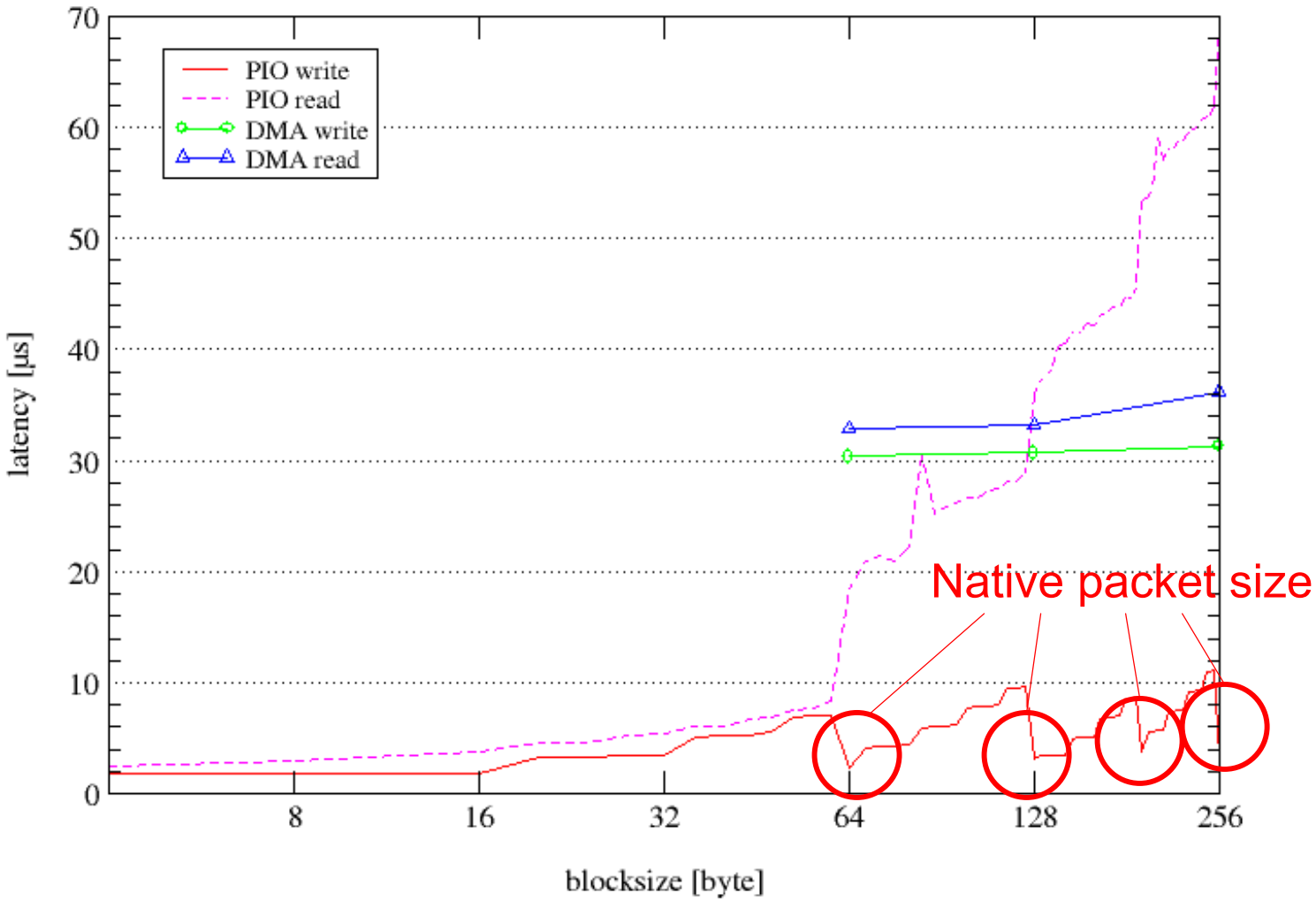
SCI – Implementation

- **Integrated Systems**
- **PCI-SCI adapter:**
  - 64-bit 66 MHz PCI-PCI Bridge
  - Key performance values on IA-32 platform:
    - Remote-write (word) 1,5 us
    - Remote-read (word) 3,2 us
    - Bandwidth PIO 170 MB/s (IA-32 PCI is the limit)
      (peak bandwidth for 512 byte blocksize)
      DMA 250 MB/s (DMA engine is the limit)
    - Synchronization ~ 15 us
    - Remote interrupt ~ 30 us

PCI-SCI – Peformance on IA-32: **Bandwidth**



PCI-SCI – Peformance on IA-32: **Latency**



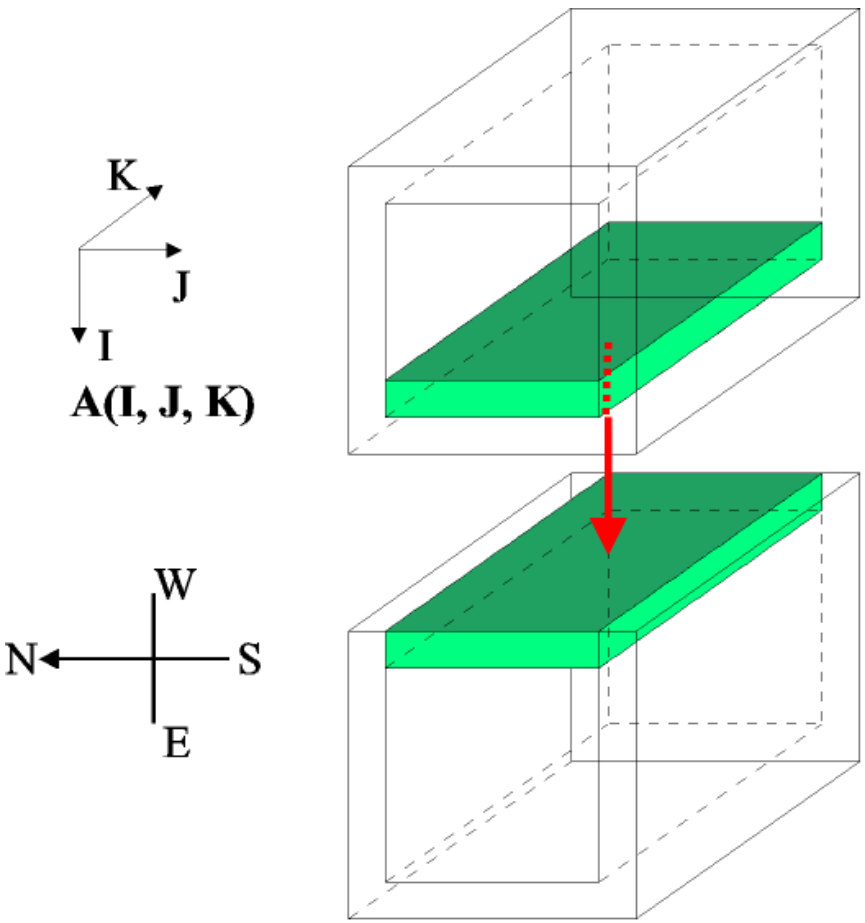
Message Passing over SCI

Similar to local shared memory, but:

- Different performance characteristics
  - Read / write latency
  - I/O-bus behaves differently than system bus
  - Granularity and contiguousity of access important
  - Load on independant inter-node links (collective operations)
- Different memory consistency model
- Connecting / Mapping incurs more overhead
- Resources are limited
- It‘s still a network: connection monitoring

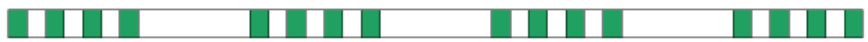
⇨ Naive approach **„map everything and do shmem“** is neither **efficient** nor **scalable**

Communicating Non-Contiguous Data



North-South boundary exchange: single strided

East-west boundary exchange: double strided



MPI Datatypes

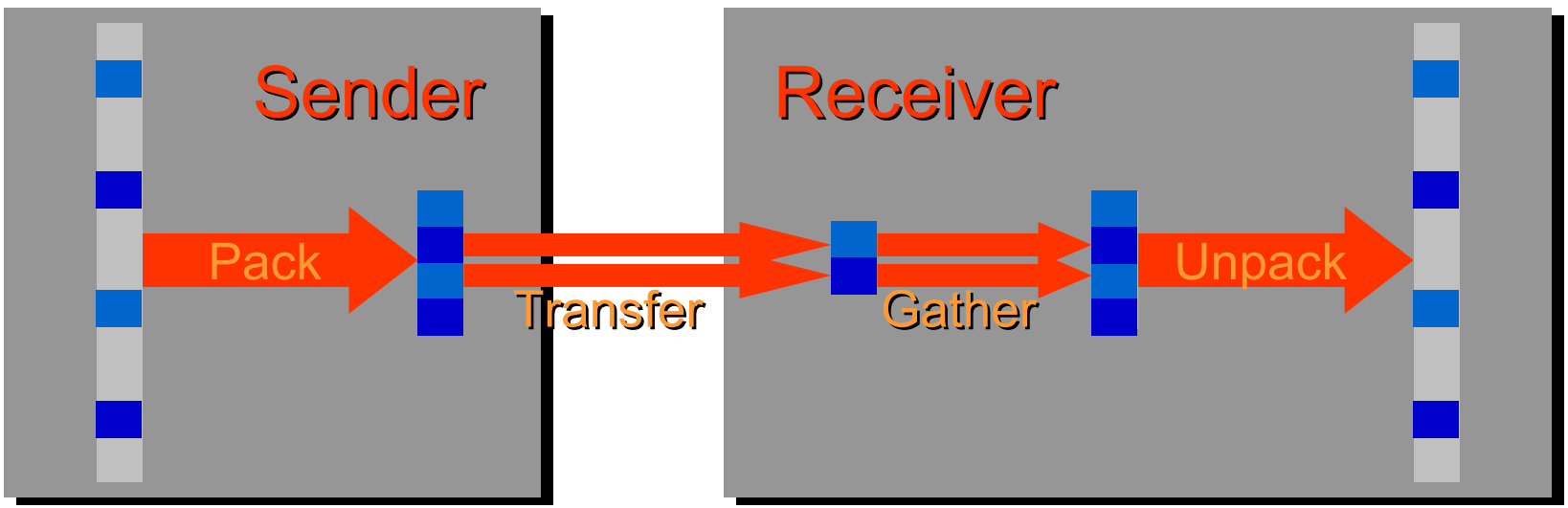
- **Basic datatypes (`char`, `double`, ...)**
- **Derived datatypes**
  - Concanate elements: communicat *vectors*
  - Associations of elements: communicate *structs*
- **Using offsets, strides and upper/lower bounds, non-contiguous datatypes can be defined**
  - Example: Columns of a matrix (‚C' storage)

Sending Non-Contiguous Data – The Generic Way

What is the problem?

- Many communication devices have simple data specification `<data location, data length>`
  - ⇨ N send operations for N separate data chunks – inefficient

Generic solution:

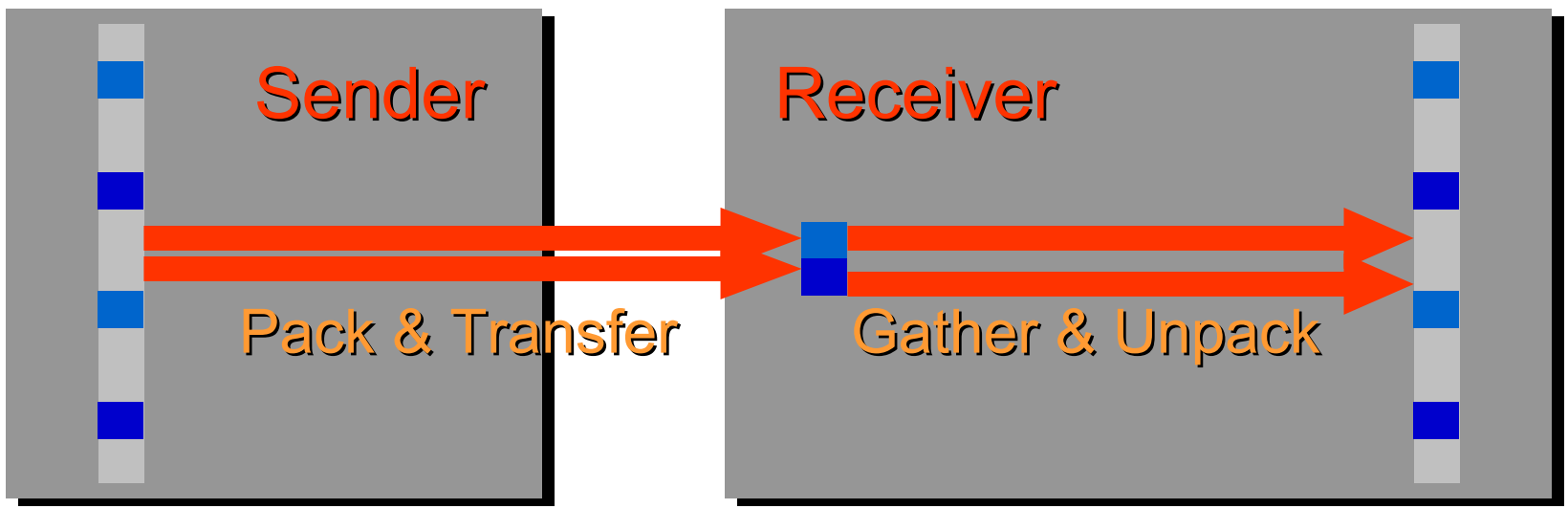


Sending Non-Contiguous Data – The SCI Way

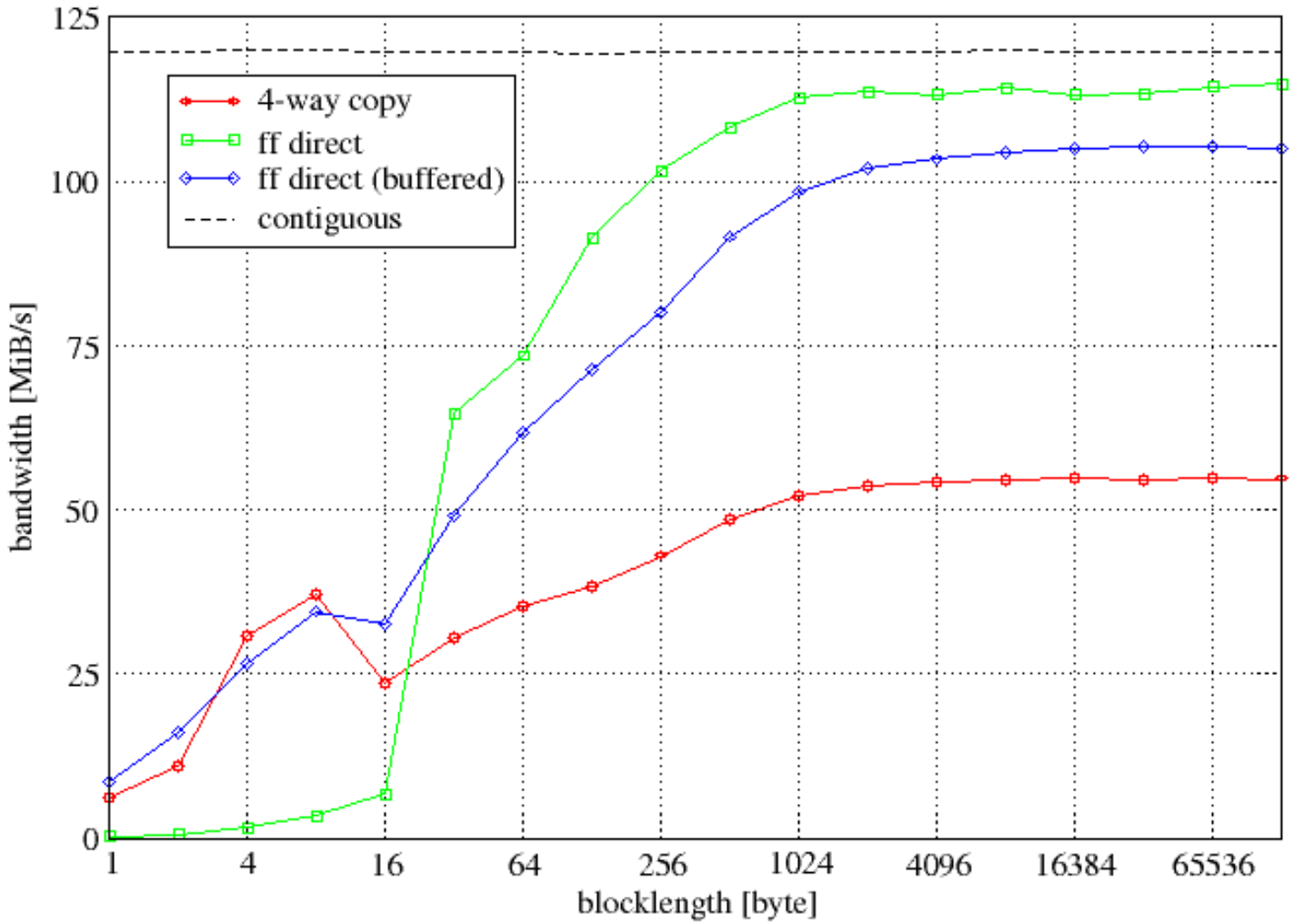
Goal: avoid intermediate copy operations!

SCI Advantage: high bandwidth for small data chunks

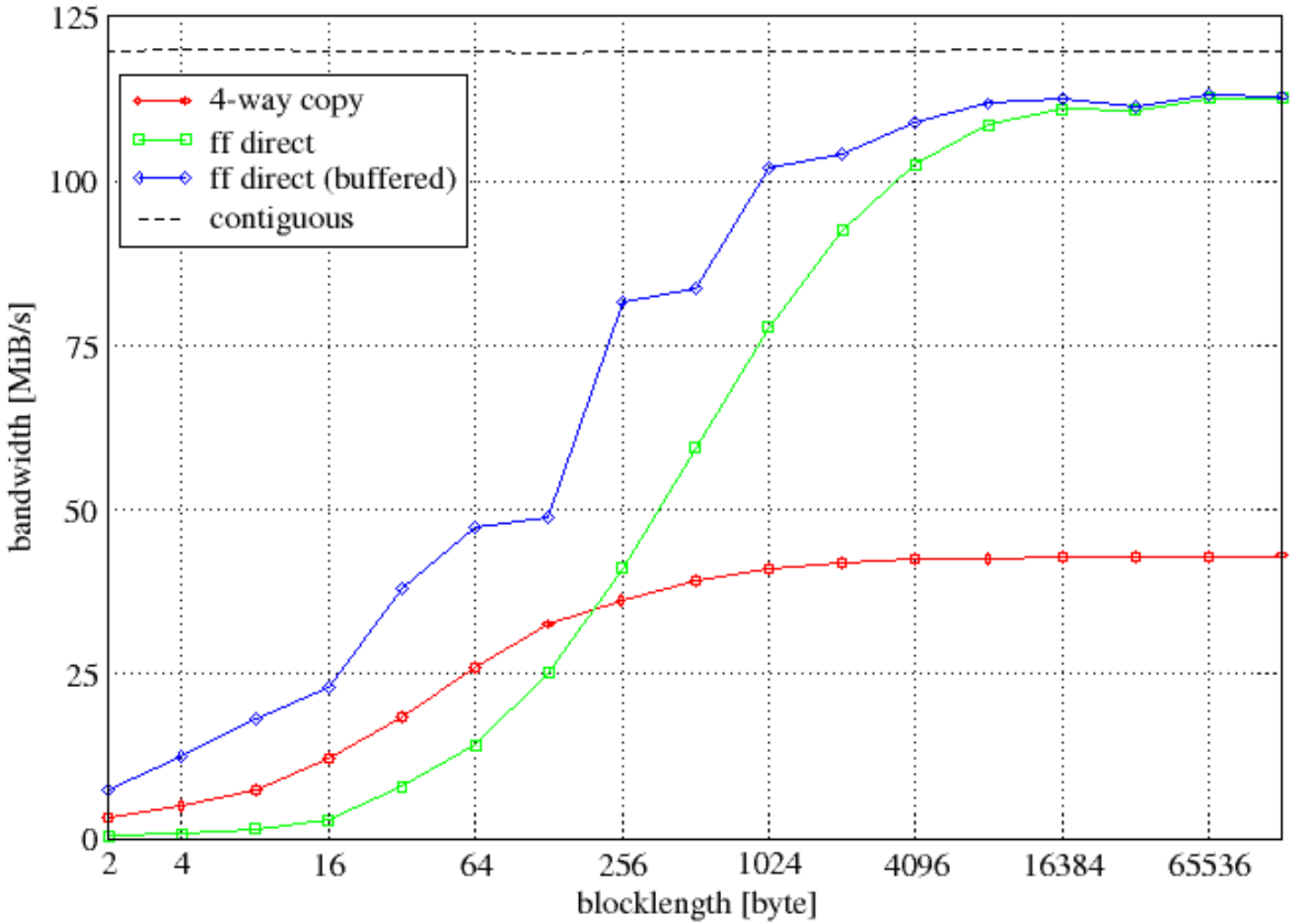
Problem: requires fast, space-efficient, **restartable** pack/unpack algorithm: **direct_pack_ff**



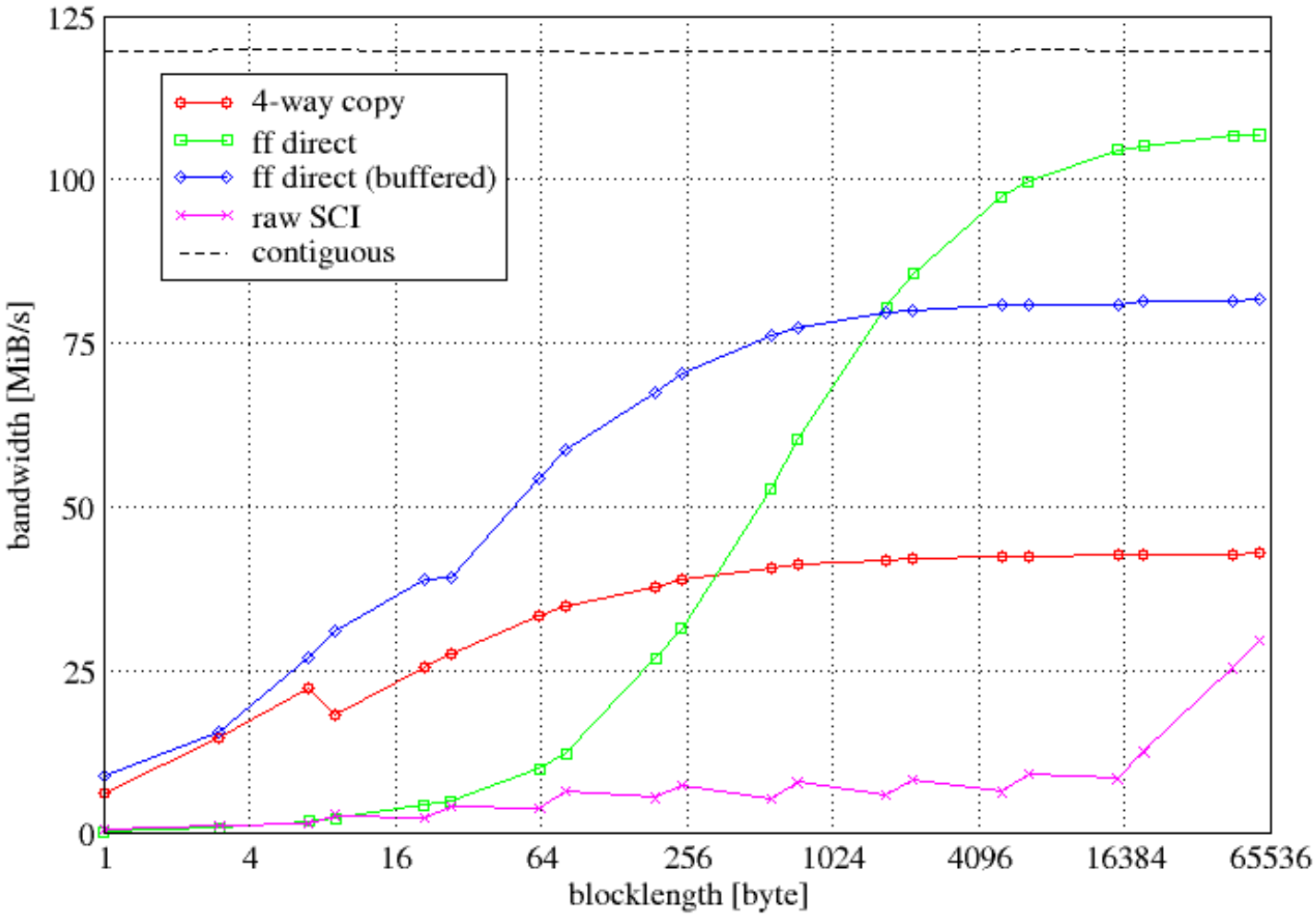
Performance (I) – **Simple Vector**



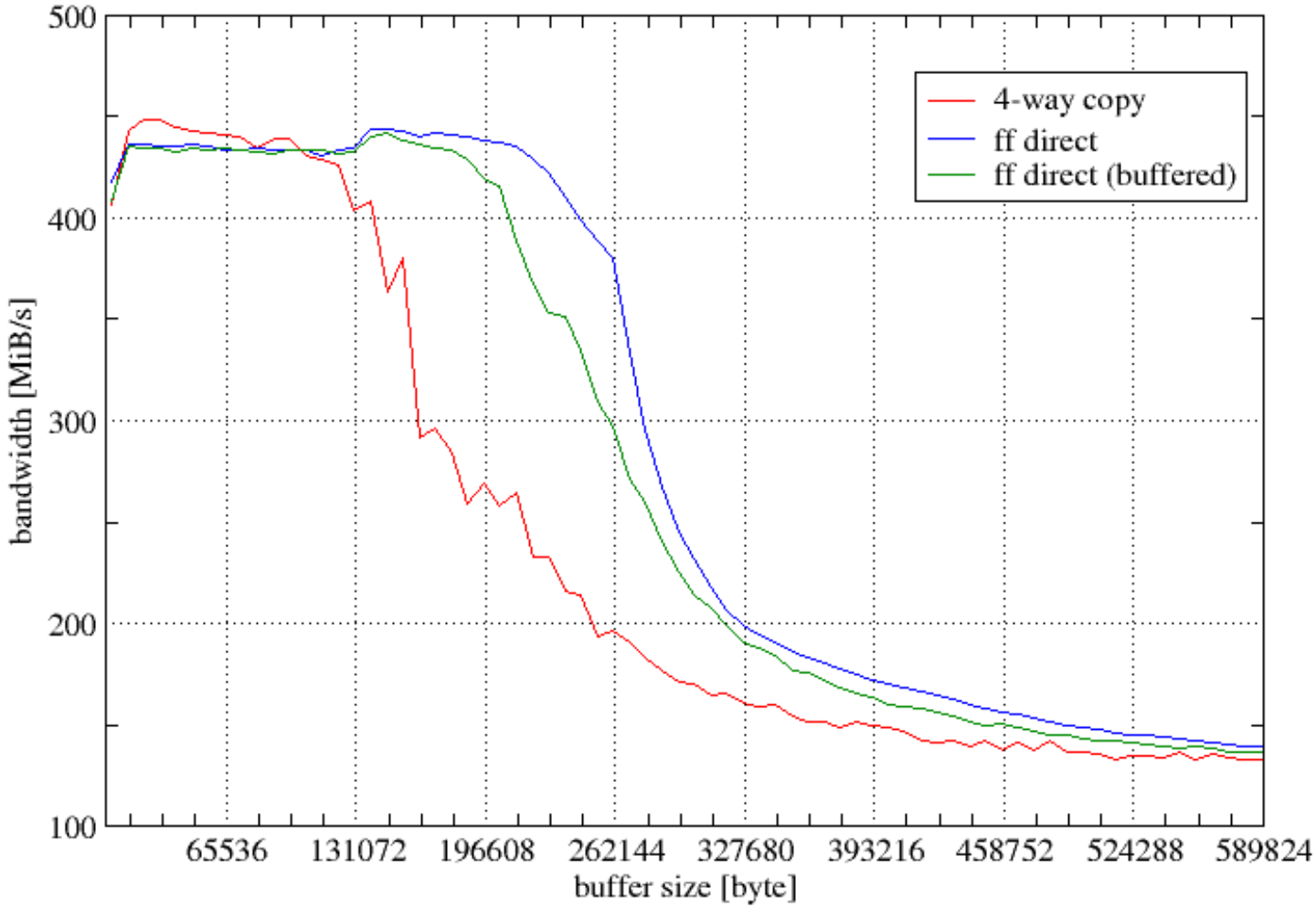
Performance (II) – **Complex Type**



Performance (III) - **Alignment**



Reduced Cache-Pollution



One-Sided Communication

SCI *is* one-sided communication, *but*:

- Remote read bandwidth < 10 MB/s
- Only fragments of each process‘ address space are accessable
- These fragments need to be allocated specifically (until now)

⇨ **Multi-protocol approach required to**

- Handle every kind of memory type
- Achieve optimal performance for each type of access

Different Access Modes

Direct access of remote memory:

- Window is allocated from an SCI shared segment
- **`Put`**-operations of any size
- **`Get`**-operations sized up to a threshold

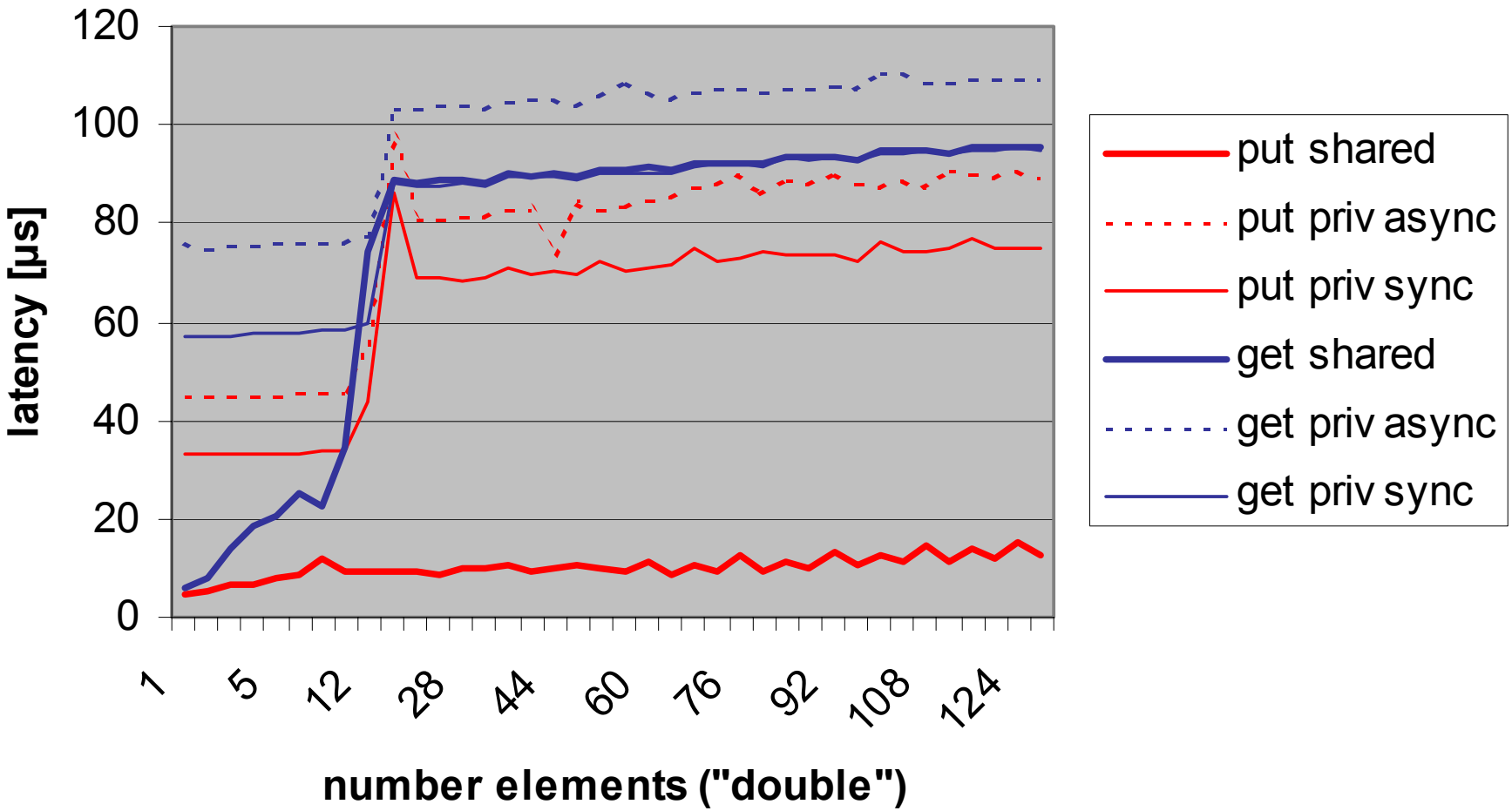
Emulated access of remote memory: Access initiated by origin via messages, executed by target:

- Windows allocated from private memory
- **`Get`**-Operations beyond threshold
- **`Accumulate`**-operations

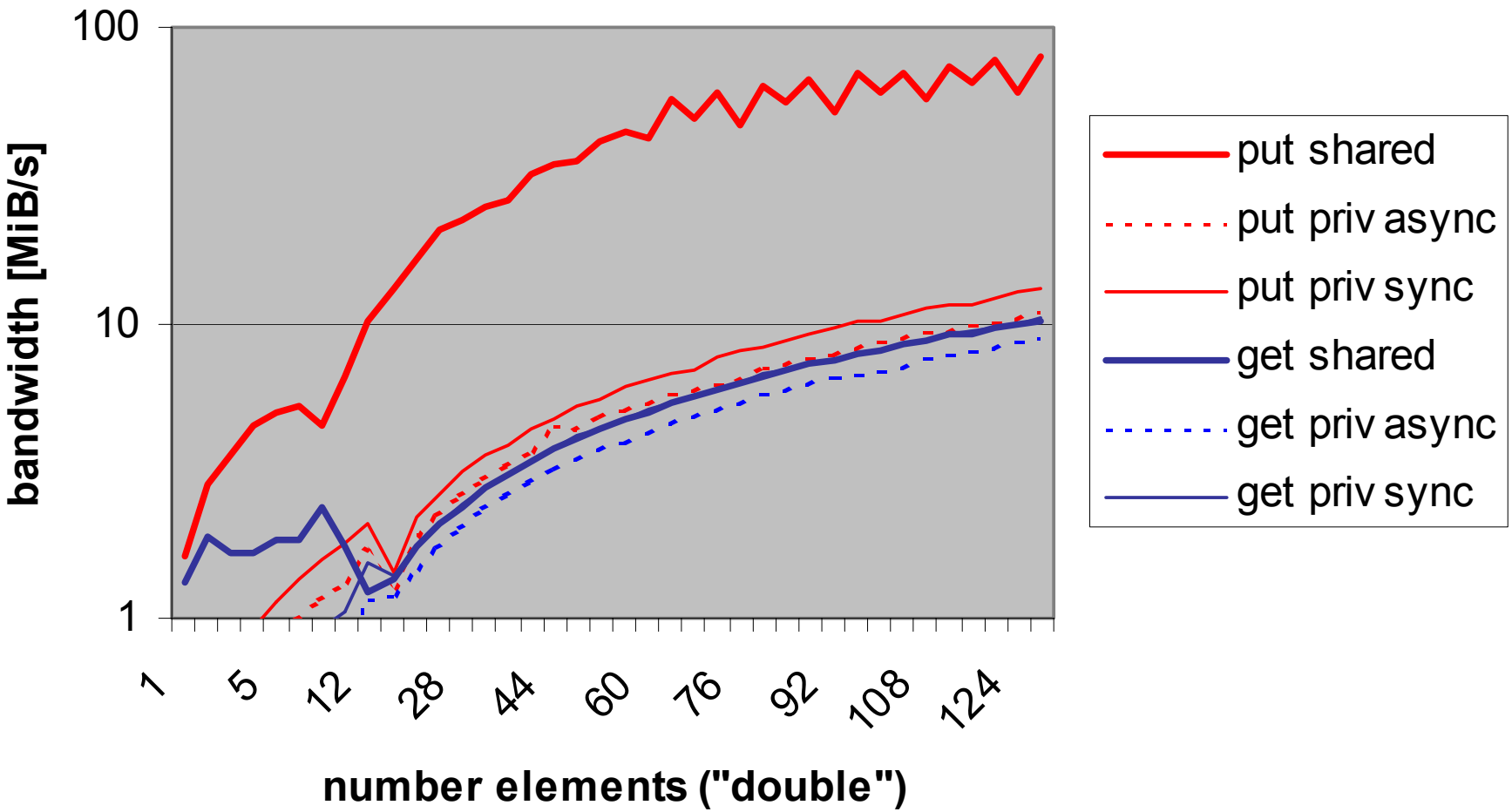
Synchronous or asynchronous completion

⇨ Different overhead & synchronization semantics

Latency of Remote Accesses



Bandwidth of Remote Accesses



Scalability

- SCI ringlet bandwidth: 633 MiB/s
- Outgoing peak traffic for `MPI_Put()` per node: 122 MiB/s
- Worst case scalability:

nodes	**offered load**	**per node** [MiB/s]	**total** [MiB/s]	**efficiency**
4	76,3 %	120,7	482,8	76,3 %
5	95,3 %	115,8	579,0	91,5%
6	114,4 %	97,8	586,5	92,7 %
7	133,5 %	79,3	555,1	87,7 %
8	152,5 %	62,8	502,2	79,3 %

Conclusions

- **Avoiding separate pack/unpack for non-contiguous datatypes**
  - Highly efficient communication
  - Reduced cache pollution and working set size
- **Performance of one-sided communication comparable to integrated systems**
  - Multi-protocol approach ensures unlimited usability with optimal performance for each scenario
- **Techniques are directly usable for SCI and local shmem**
- **Limits** (and possible solutions)**:**
  - Address space of IA-32 architecture (IA-64 or others)
  - Limited address translation resources (better hw, or sw cache)
  - Ringlet scalability (increase link clock)
  - Remote write latency (PCI-X w/ 133MHz -> down to 0.8 µs)
  - Remote read-bandwidth (no real solution in sight)
  - Interrupt latency (interrupt-on-write will halve it)

Thank you!

Spare Slides

Pack on the fly

Store type information on ‚commit' of datatype:

offset, length, stride, repetition count

- Usual data structure **tree:** *recursive, complex restart*
- Naive data structure **list**: *not space-efficient*
- Suitable data structure: **linked list of stacks**

- Bandwidth break-even point between
  - Overhead for intermediate-copies
  - Reduced bandwidth for fine-grained copy operations

  ⇨ List **sorted by size** of stack entries („leaves") to allow for optimization: **buffering for small and misaligned (parts of) leaves**

Integration in SCI-MPICH

Building the stack:

```
MPIR_build_ff (struct MPIR_DATATYPE *type,
  int *leaves)
```

Moving data:

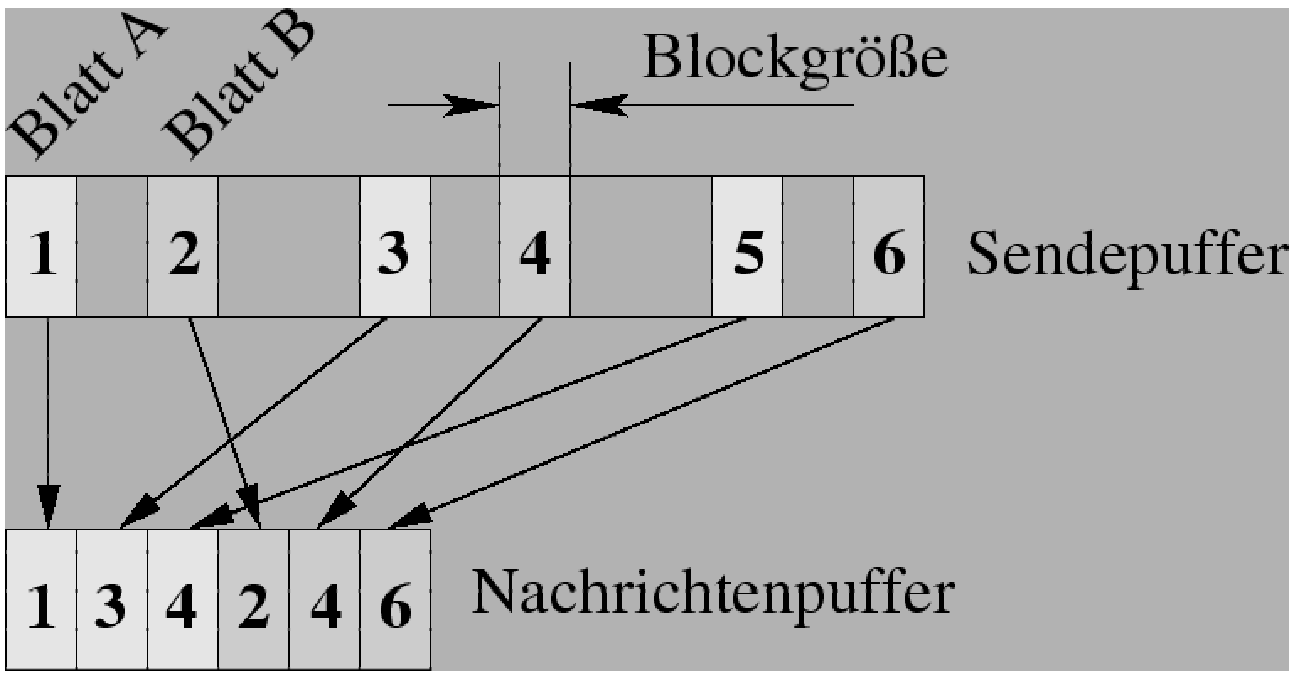
```
MPID_SMI_Pack_ff (char *inbuf,
  struct MPIR_DATATYPE *dtype_ptr,
  char *outbuf, int dest, int max, int *outlen)

MPID_SMI_UnPack_ff (char *inbuf,
  struct MPIR_DATATYPE *dtype_ptr,
  char *outbuf, int from, int max, int *outlen)
```

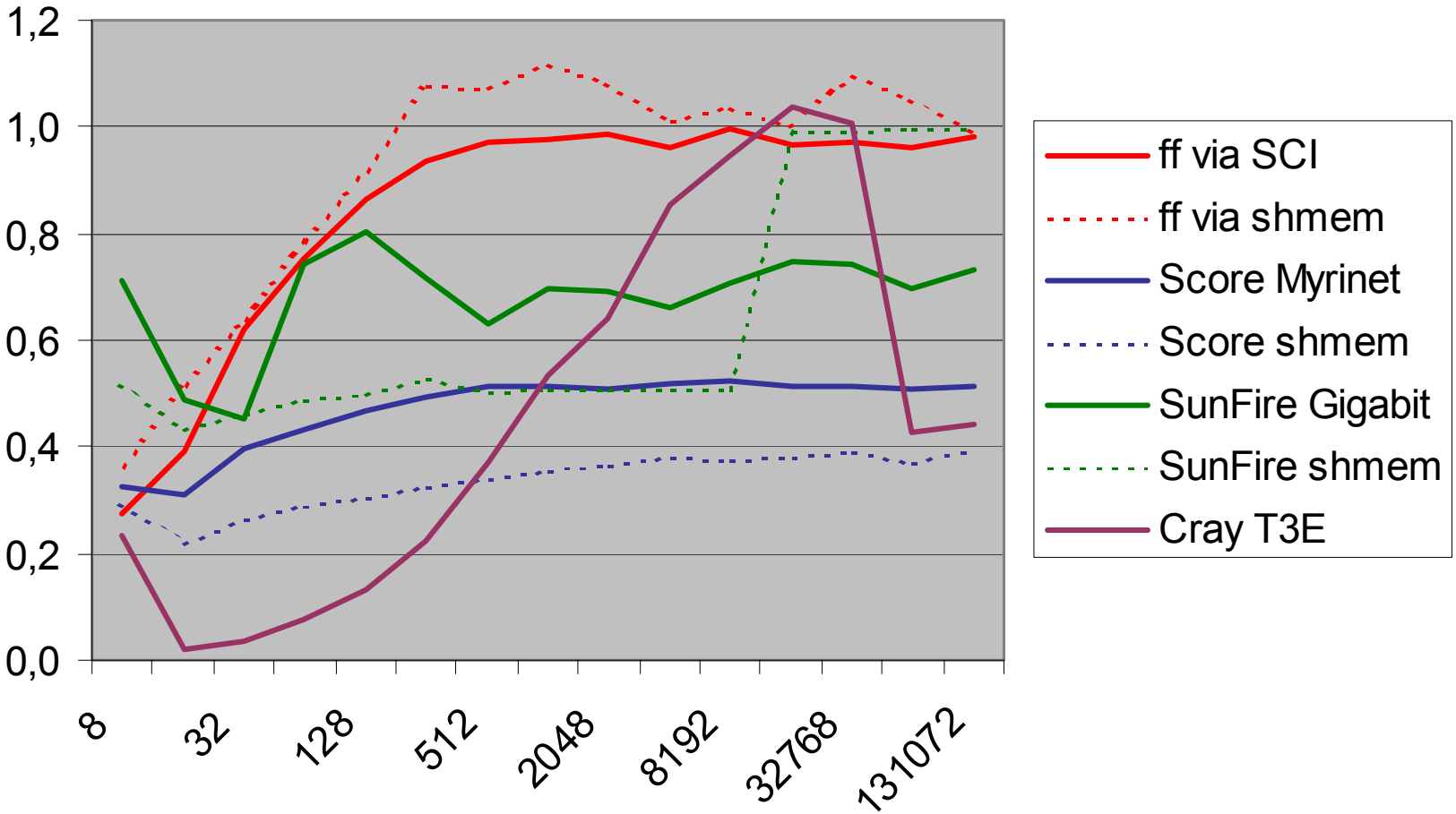
Type Matching

Sorting of basic datatypes (*leafs*) by size:

- Improves performance by mixing gathering & direct write
- Changes order in incoming buffer at receiver!



Efficiency Comparision: Non-contig Vector



Memory Allocation

`MPI_Alloc_mem()`:

- Below threshold 1: allocate via malloc()
- Below threshold 2: allocate from shared mem pool
- Beyond threshold 2: create specific shared memory

Attributes:

- Keys `private`, `shared`, `pinned`: type of memory buffer
- Key `align` with value: align memory buffer

`MPI_Free_mem()`:

- Unpin memory, release shared memory segment

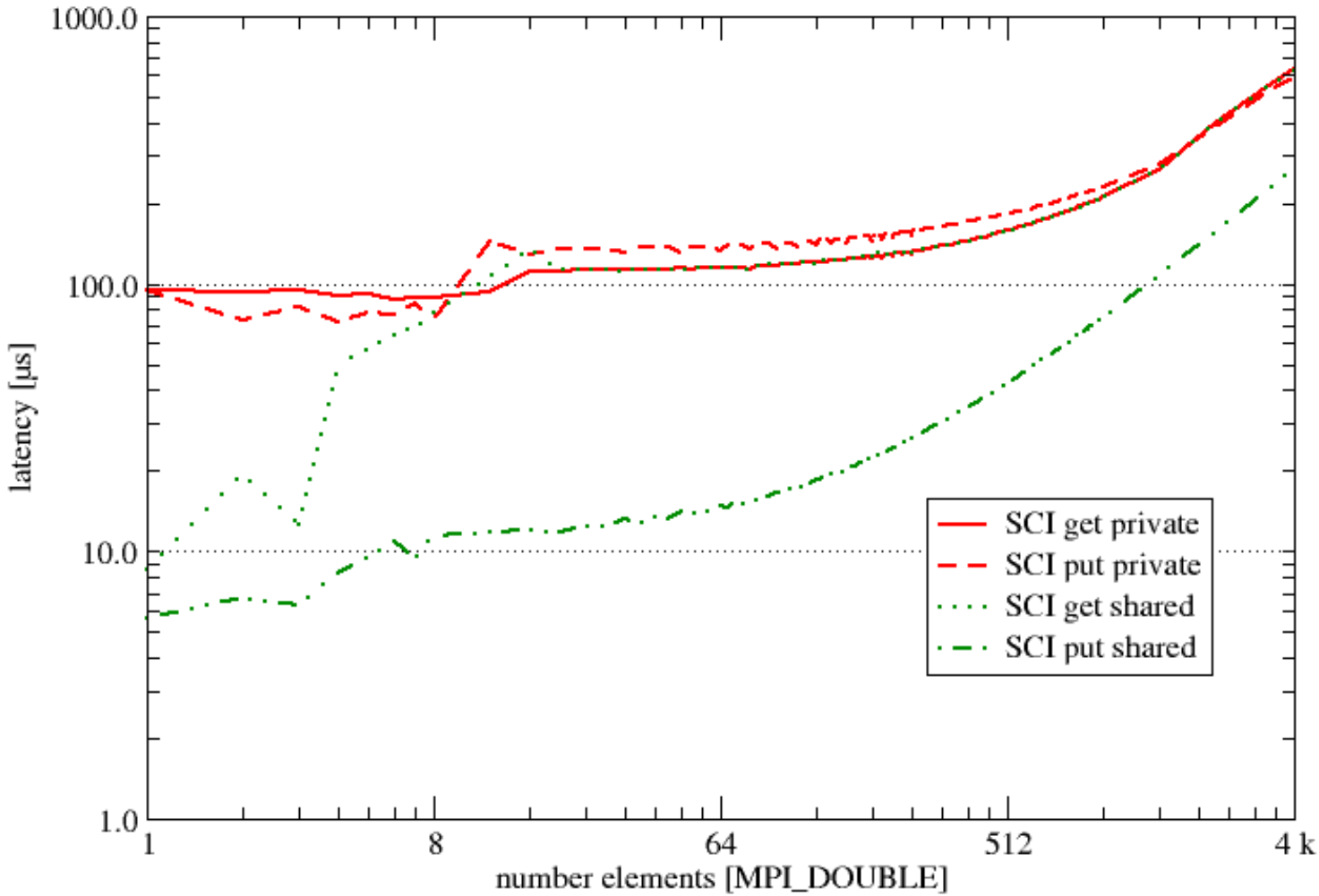
⇨ Implicitely invoke *remote segment callbacks*

sparse micro-Benchmark

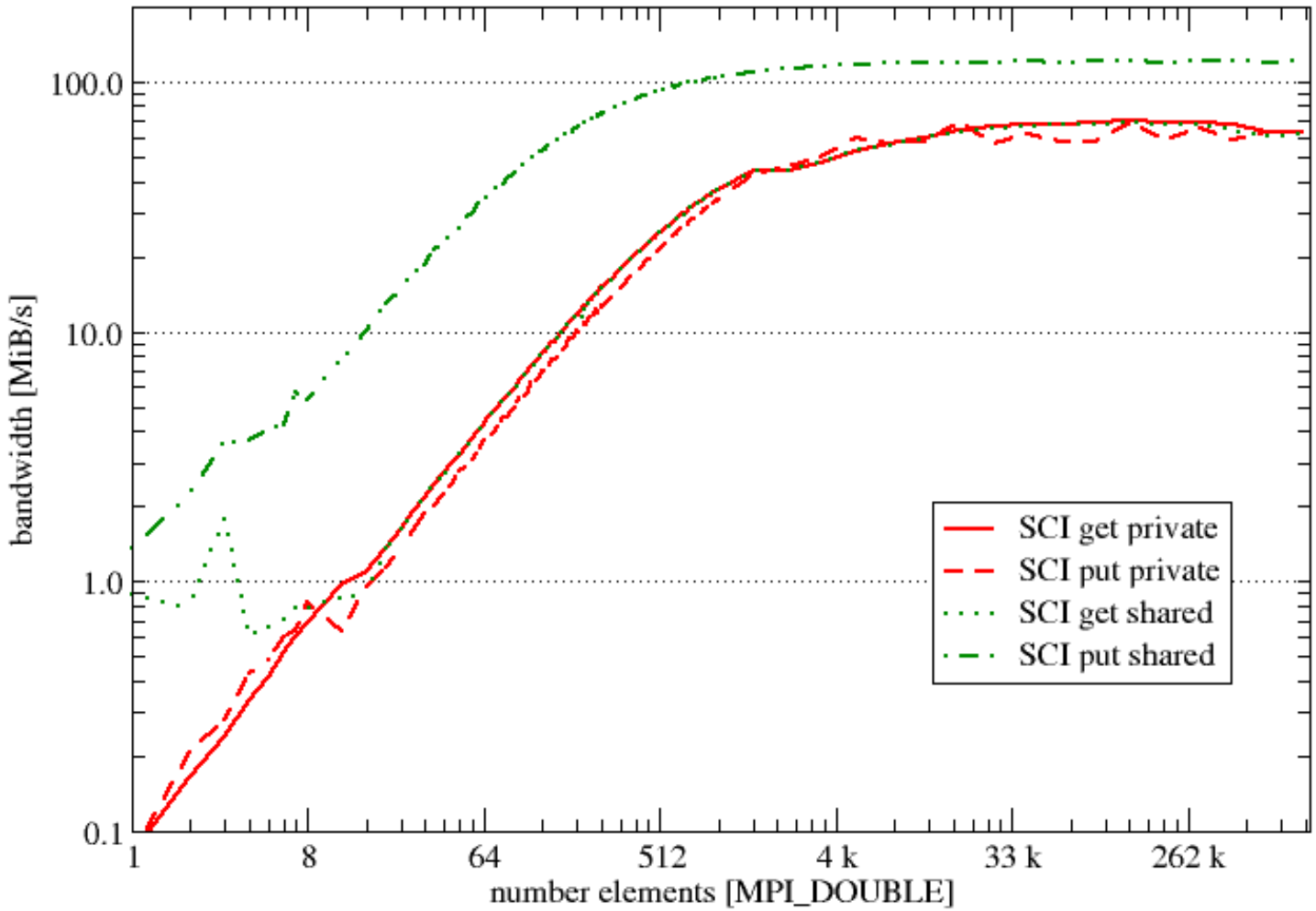
Simulate access to remote sparse matrix:

```
MPI_Win_create (..., winsize, ...)
for (increasing values of access_cnt) {
   offset = 0
   stride = access_cnt * sizeof(datatype)
   flush_cache()
   time = MPI_Wtime()
   while (offset + access_size < winsize) {
      MPI_Get/Put (.., partner, offset, access_cnt, ..)
      offset = offset + stride
   }
   MPI_Win_fence(...)
   time = MPI_Wtime() - time
}
```

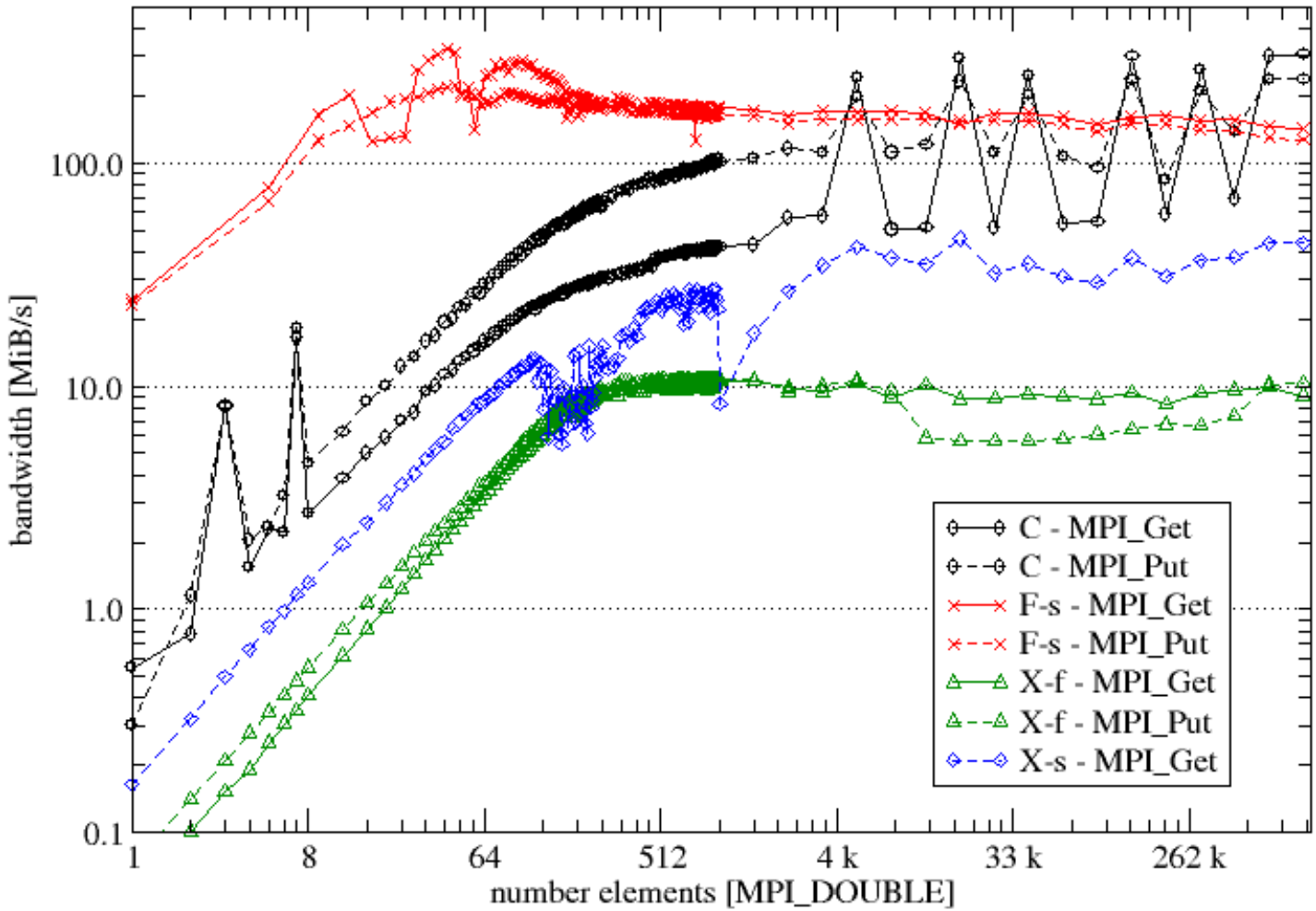
Latency of Remote Accesses



Bandwidth of Remote Accesses



Point-to-Point Comparison



Scaling Comparison `MPI_Put`