CRAY®

EMEA RESEARCH LAB

Optimisation of Data Movement in Complex Workflows

Tim Dykes, Aniello Esposito, Clement Foyer, Utz-Uwe Haus,
Harvey Richardson, Karthee Sivalingam, Adrian Tate

18th Workshop on High Performance Computing in Meteorology

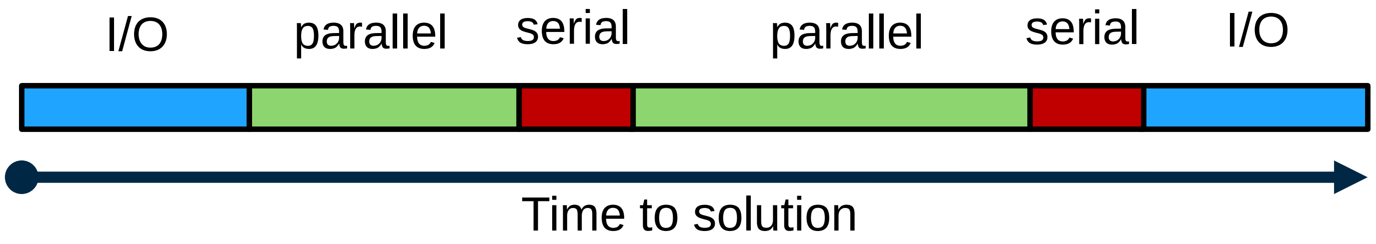
Agenda

- Challenges observed by us and customers
- The Octopus project
- The MAESTRO and EPIGRAM-HS projects
- Universal Data Junction (UDJ) and data redistribution
- A use-case (WRF visualization)

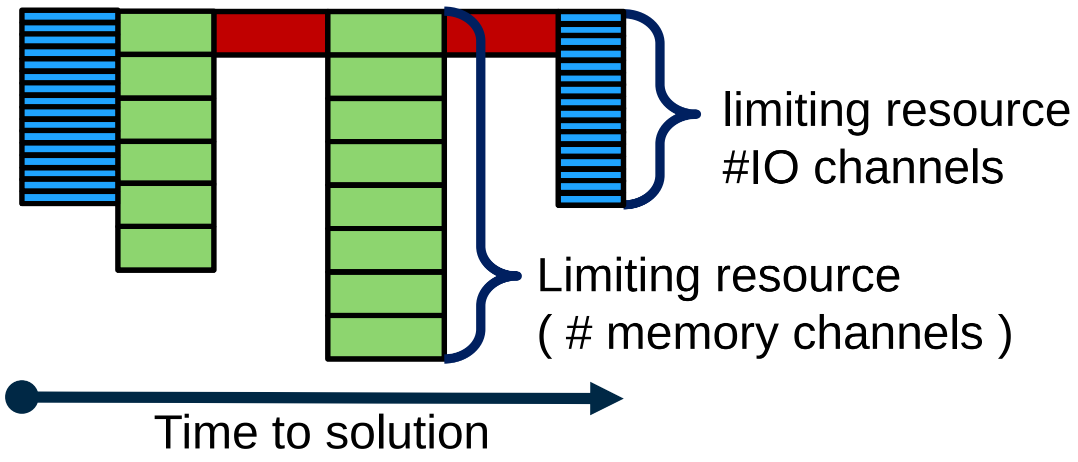
Complaints we hear from (tetchy) customers

- **Software stack remains ill-suited for modern systems (getting worse)**
  - Why are we still using a programming environment designed in the age of FLOPS?
  - Where's my consistent data interface?
- **So-called "next-gen memory hierarchy" never showed up**
  - Not actually a hierarchy…
  - Where is HBM for CPUs?
  - How will we use use NVRAM?
  - Where's the working memory model?
- **IO interfaces are less than useful**
  - We hate POSIX but there's still nothing better
- **You're focusing on a small piece of the pie**
  - Only a small piece of the scientific workflow has been treated well
  - Simulation (e.g. CFD) done well but analysis, post-processing, usage models are ignored
  - <u>Time-to-solution is very important, but not the only game in town</u>

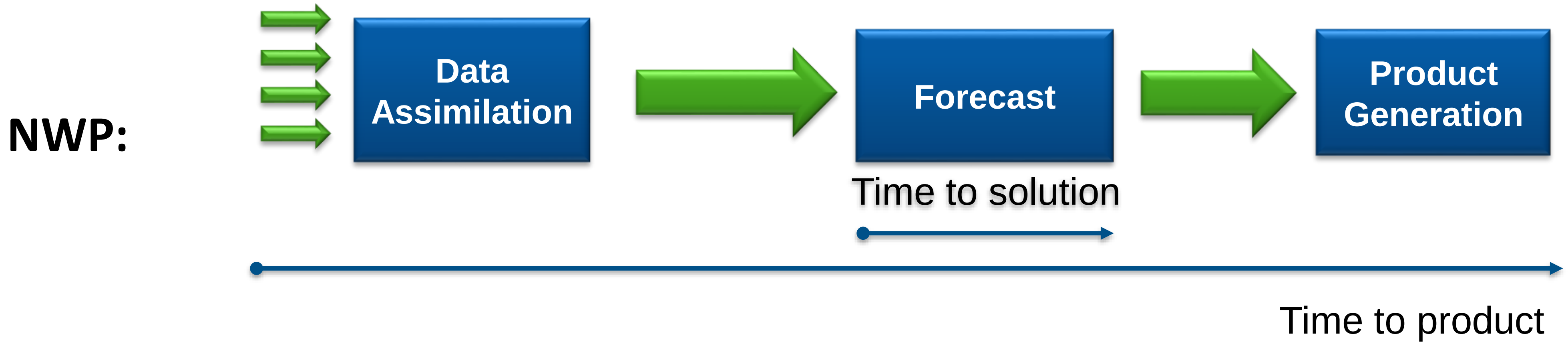
Time to solution



- Many forms of parallelism
- Algorithmic advances
- Code optimization (Compiler & hand)
- ISA features
- Programming Models
- Performance Abstraction
- Systems Software / Operational
- Network, memory increases

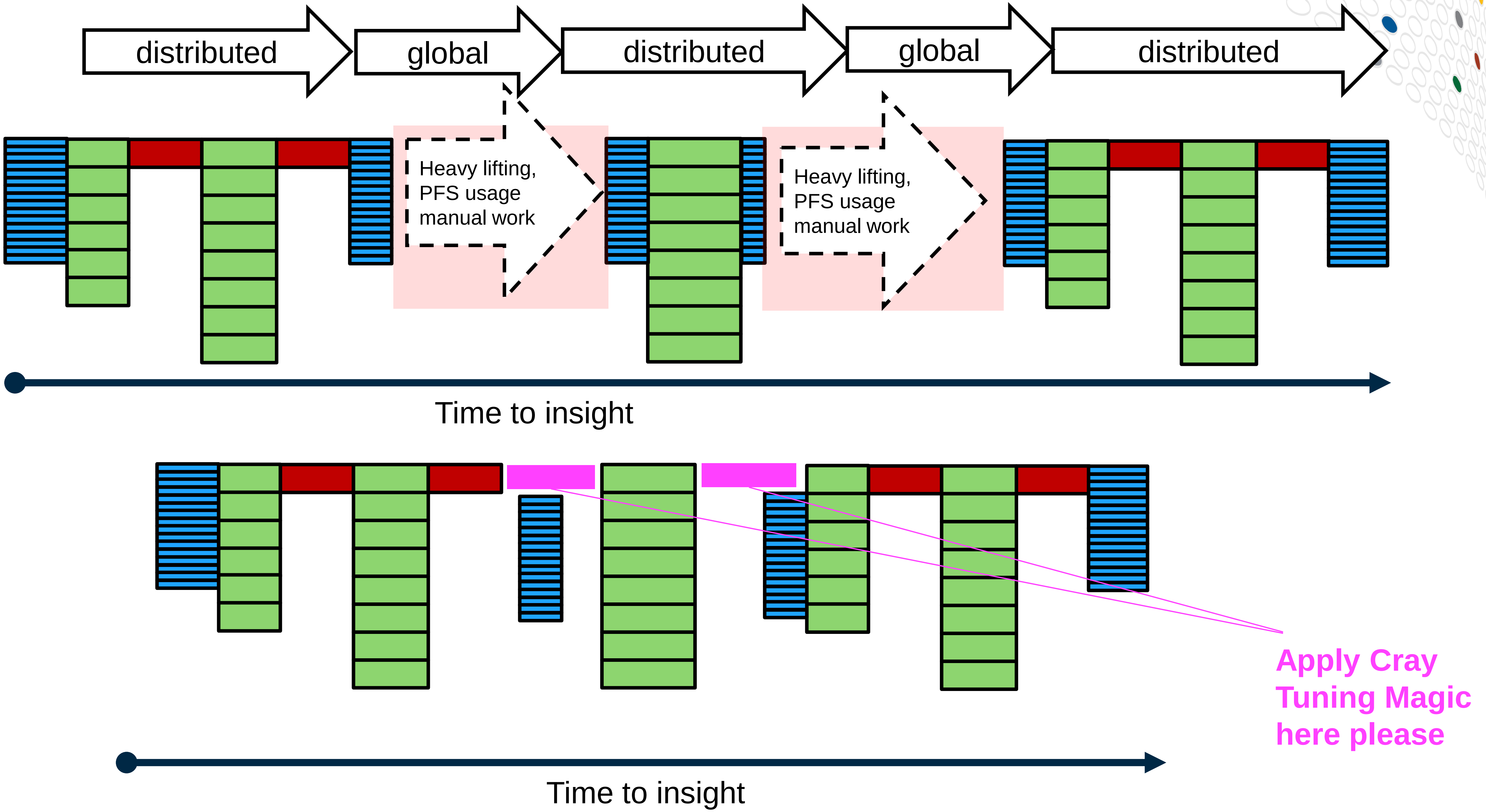


Time to scientific product / insight



What does ➡ mean?

How do we optimise ➡ to be smaller?


distributed
global
distributed
global
distributed
Heavy lifting,
PFS usage
manual work
Heavy lifting,
PFS usage
manual work
Time to insight
Apply Cray
Tuning Magic
here please
Time to insight

1. Data-centric view of workflows

2. Parallel Data Handling and re-distribution

3. Object-like and transaction interface to user-data

4. Minimally Invasive API at multiple levels app, systems software)

5. Pragmatic Model of Memory System

6. Interface to all memory and storage

7. Resource-aware adaptive Transport

8. Minimization of data movement

New EU H2020-FETHPC-2017 Projects

EPIGRAM-HS: Exascale ProGRAmming Models for Heterogenous Systems

Maestro: Middleware for memory and data-awareness in workflows

EPIGRAM-HS

EPiGRAM-HS is developing a **programming environment,** enabling HPC and emerging **applications** to run on large-scale heterogeneous systems at maximum performance

Network | Memory | Compute

Applications

EPIGRAM-HS Applications

Traditional HPC Applications

- IFS – Weather Forecast – ECMWF
- Nek5000 – CFD – KTH PDC
- iPIC3D – Space Physics – KTH PDC

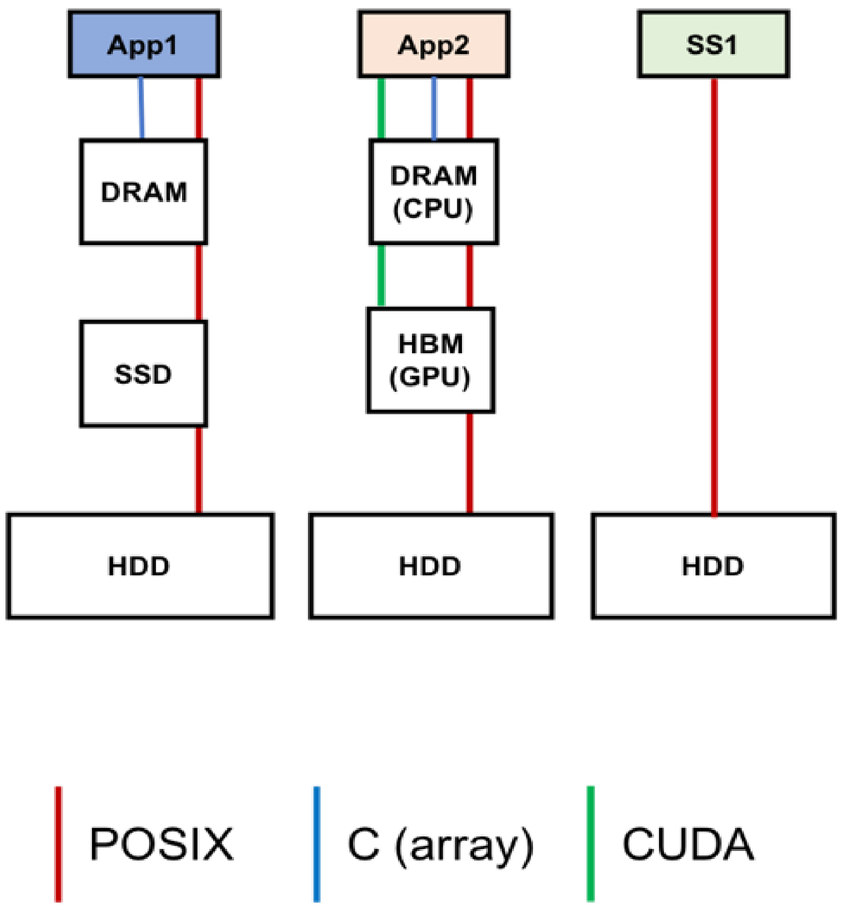
Emerging AI Applications

- Lung Cancer Detection – Caffe / TensorFlow – Fraunhofer
- Malware Detection – Caffe / TensorFlow – Fraunhofer

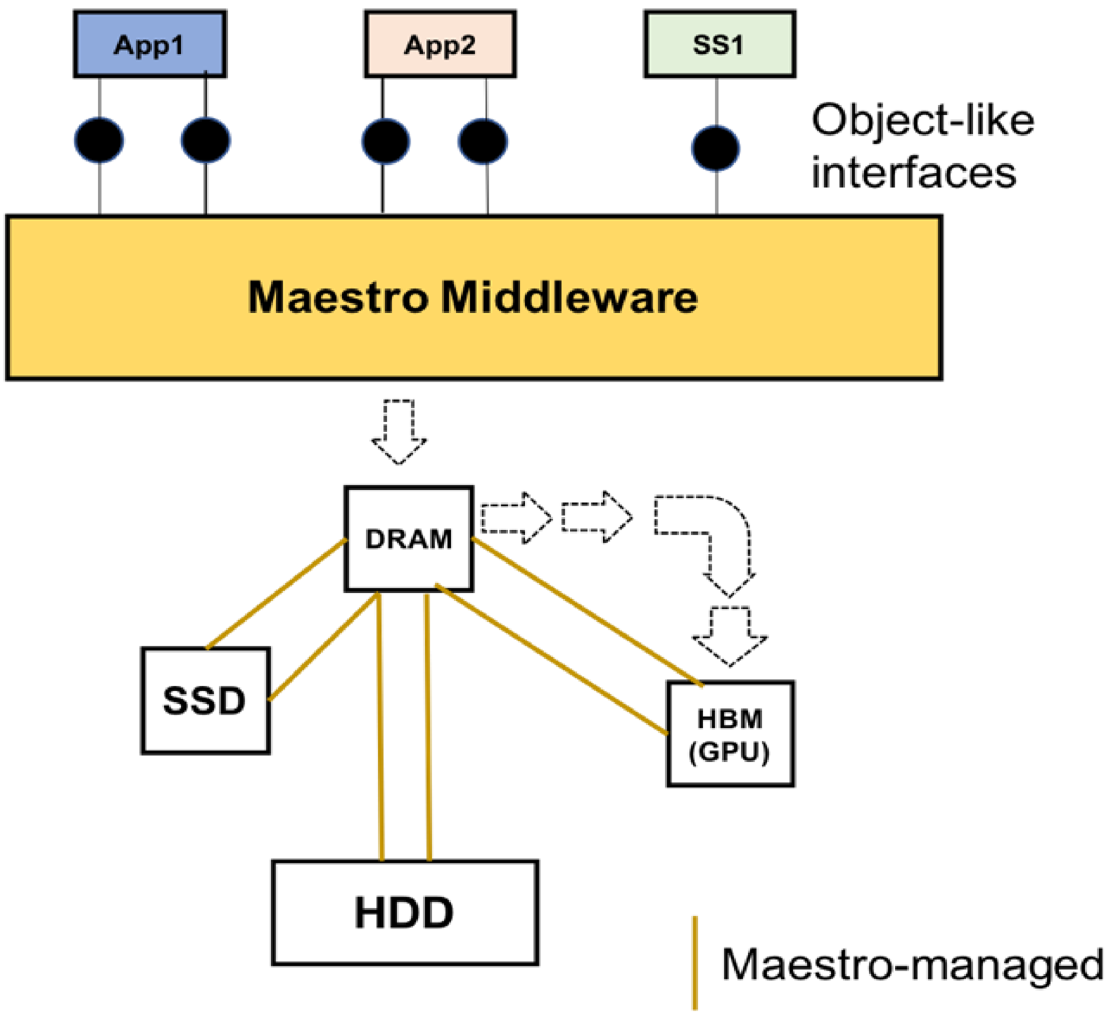
Maestro Project

- **FETHPC-2017 Consortium**
  - Industrial partners
    - CRAY (Switzerland), Seagate (UK)
    - Research organisations / supercomputing centres
    - CEA (France), CSCS (Switzerland), ECMWF (international), JSC (Germany)
  - SME
    - Appentra (Spain)
- **Goals**
  - Develop a middleware providing consistent data semantics to multiple layers of the stack
  - Demonstrate progress for applications through memory- and data-aware (MADA) orchestration
  - Enable and demonstrate next-generation systems software MADA features
  - Improve the ease-of-use of complex memory and storage hierarchy

Current



Maestro



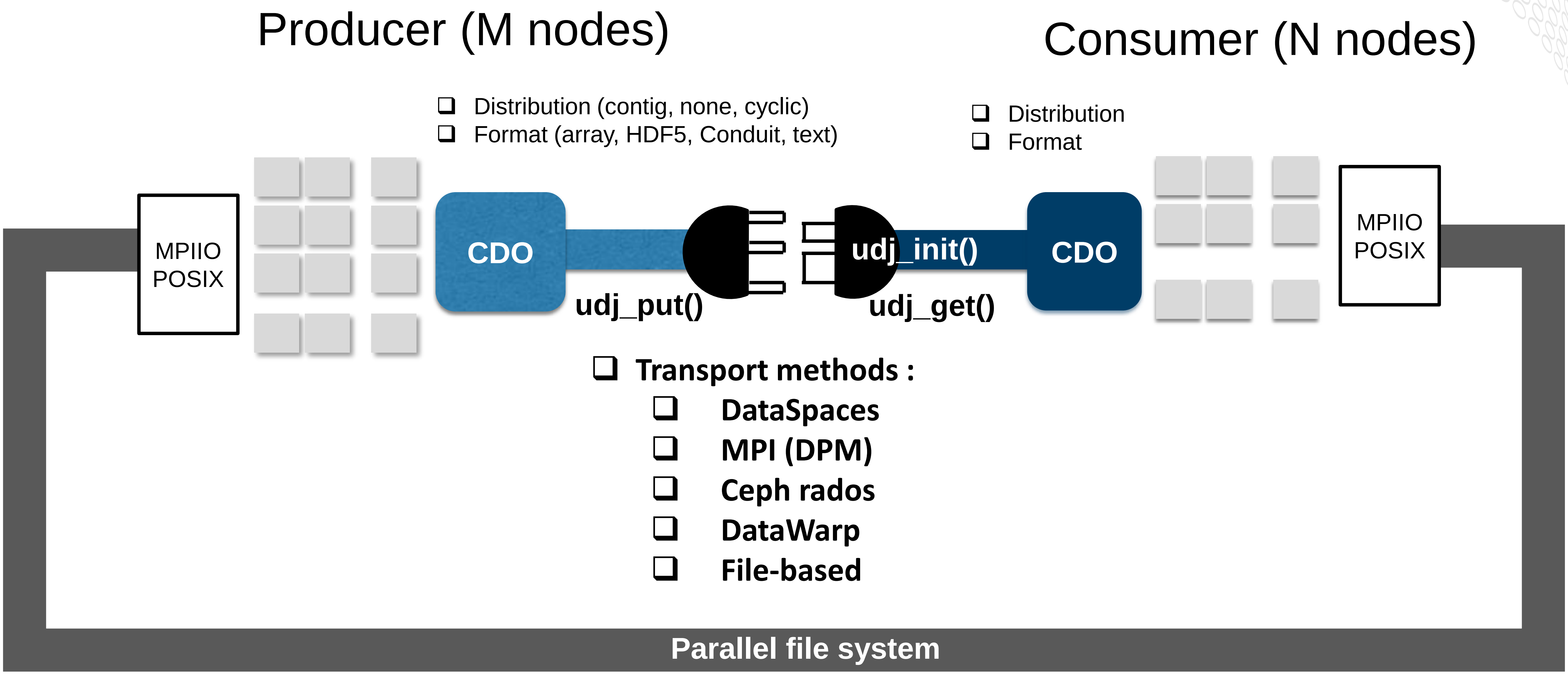
Maestro Capabilities

- A middleware library accessible from multiple levels of the stack
- Access data using an object-like and transactional interface
- Application gives over control of “Core Data Objects” to Maestro
- Maestro moves data wherever it is best placed during this time
- Gives back data to application satisfying requested data qualities

So how did we start on Octopus

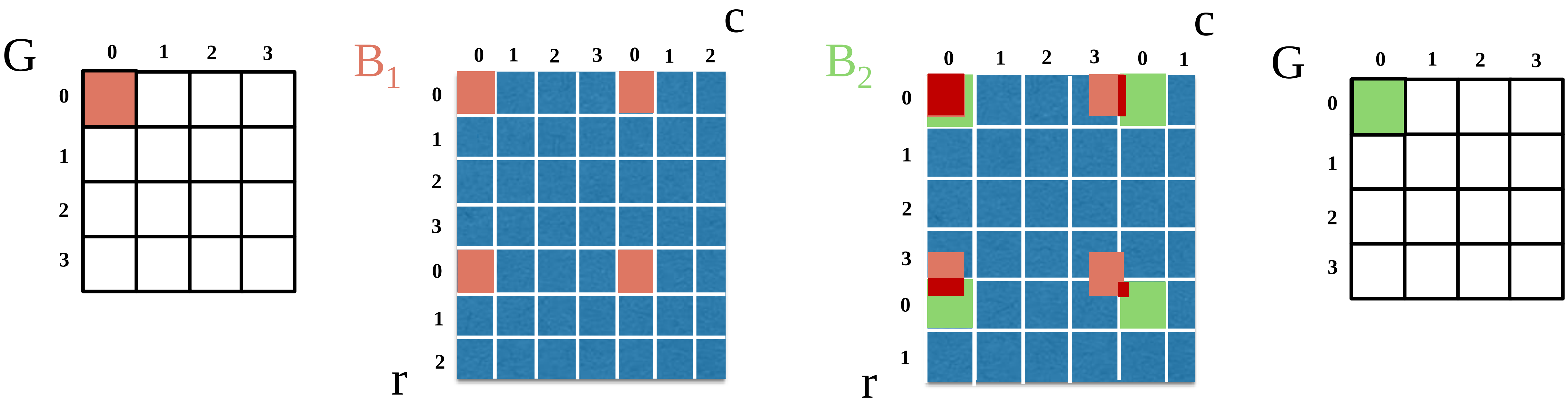
- **Workflow scheduling / tasking is a big topic (so leave that for collaborations)**
  - For HPC only this might be unrealistic anyway
- **We can implement a way to move data (UDJ)**
- **We can work with distributed data**
- **Describe (distributed) data as “objects”**

Universal Data Junction (UDJ)



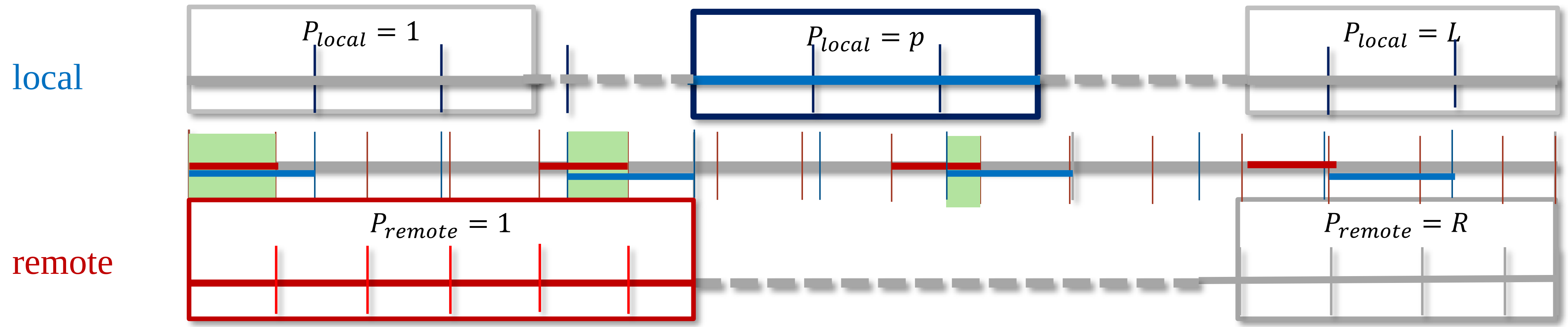
Non-triviality of Producer-Consumer Redistribution

- 2d data set dim r x c in memory
- Distributed according to some distribution scheme D_1 = (G, B_1)



- Re-distributed according to new distribution scheme D_2=(G, B_2) on same grid G
- Must communicate the non-trivial intersection data (red) for every process pair

Classical Redistribution



On each local rank:

For each d in #Dimensions

For each p in length(remote_grid(d))

For each loc in #NumLocalBlocks

For each rem in #NumRemoteBlocks

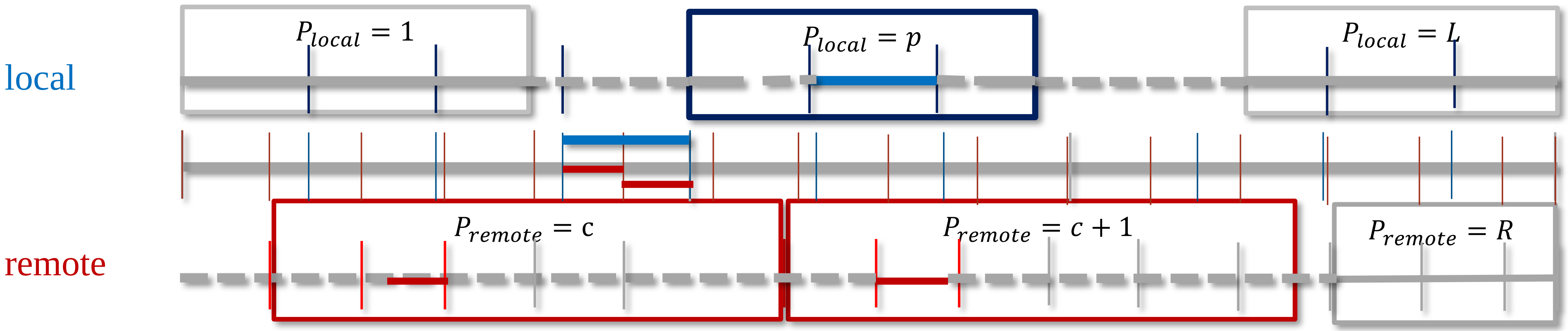
```
if MAX(loc2glob(loc),loc2glob(rem))<
        MIN(loc2glob(loc+b1),loc2glob(rem+b2)) → Add to intersection
```

Intersection = $i^1 \times i^2 \times \cdots \times i^d$

Complexity: $O(\#Dim \cdot L \cdot C \cdot n_{local} \cdot n_{remote})$

Ignores three types of periodicity!

ASPEN: Adjacent Shifting of PEriodic Node data



On each local rank:

For each d in #Dimensions

For each loc in #NumLocalBlocks

```
if (loc2glob(loc) % s2 ) <= b2  → Add to intersection
```

For each sub in b_{local}/b_{remote}

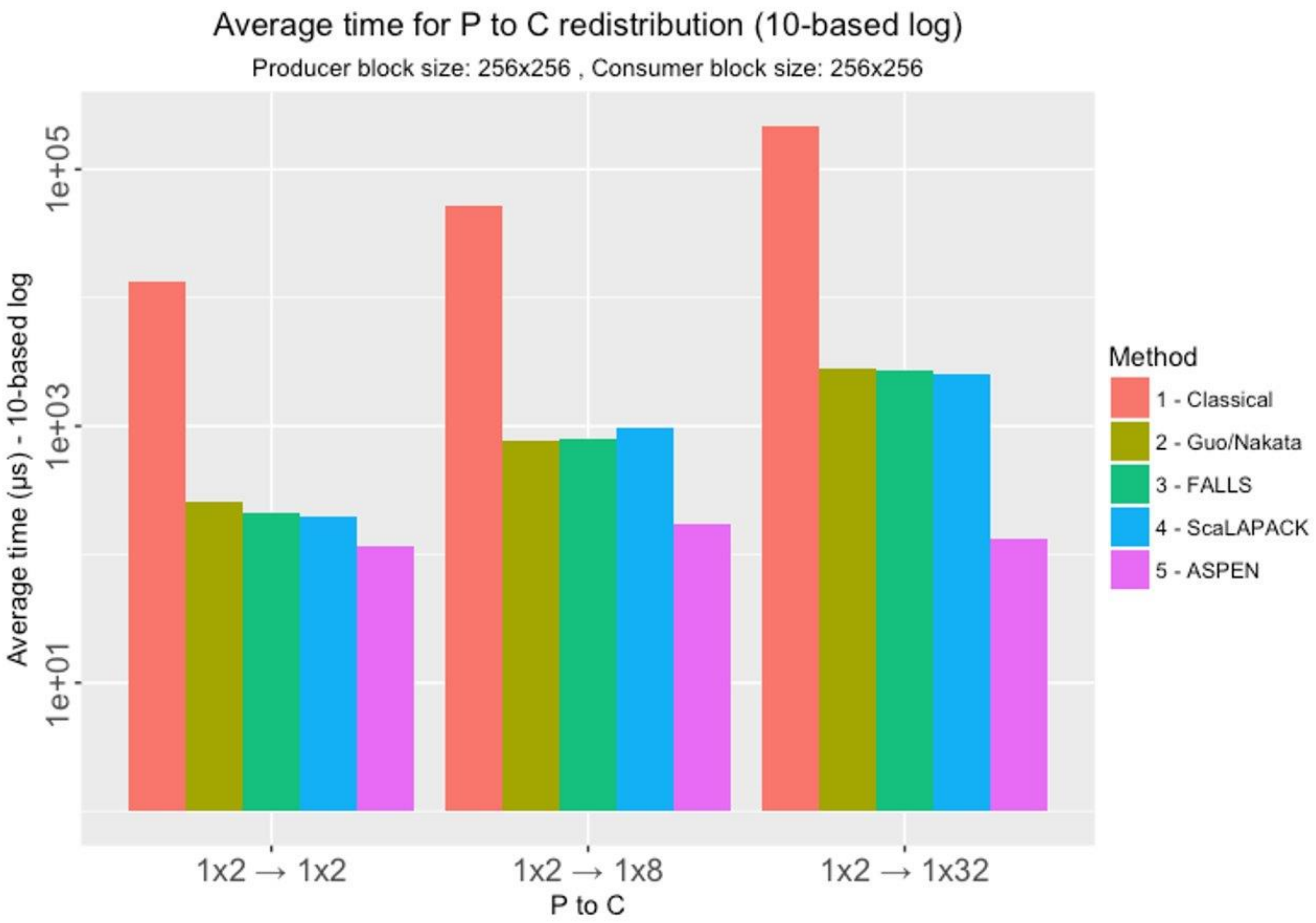
```
→ Add sub to intersection
```

Intersection = $i^1 \times i^2 \times \cdots \times i^d$

$$Complexity: O(\#Dim \cdot L \cdot \hat{n}_{local} \cdot {b_{local}}/{b_{remote}})$$

Foyer, Tate, McIntosh-Smith, ”Aspen…” in: Euro-Par 2018: Parallel Processing Workshops, Springer

Results



Average time for P to C redistribution (10-based log)
Producer block size: 256x256 , Consumer block size: 256x256
Average time (μs) - 10-based log
1e+05
1e+03
1e+01
1x2 → 1x2
1x2 → 1x8
1x2 → 1x32
P to C
Method
1 - Classical
2 - Guo/Nakata
3 - FALLS
4 - ScaLAPACK
5 - ASPEN

Using UDJ

Use and initialization:

- `#include "udj.h"`
- link with `-ludj`
- call `udj_init()`

- Define CDO views for data to be transported using UDJ
  - No data copying needed
  - Distribution description and size
    - General case
    - … and convenience methods
  - CDO ID ("Tag")
- Send/Receive as needed
  - Synchronous or asynchronous

- call `udj_finalize()`

Runtime configuration

- Set specific transport method
  - `env UDJ_TRANSPORT_ORDER=MPI,RADOS,FS`
    - Default is to automatically choose best available

Advanced usage

- ❑ Use multiple transports explicitly
- ❑ Use scripting language interface
  - ❑ SWIG wrappers for python for `udj.h`

Integrating UDJ into an existing application: MPI-IO

Producer

```
/* SPMD MPI-IO write/read coupling */
double Matrix[dim1][dim2]; /* on each rank */
…
my_offset = MYRANK*dim1*dim2*sizeof(double);
MPI_File_open(MPI_COMM_WORLD, filename,…,&fh);
MPI_File_seek(fh, my_offset, MPI_SEEK_SET);
MPI_File_get_position(fh, &my_current_offset);
MPI_File_write(fh, &Array, dim1*dim2,
MPI_DOUBLE,…);
MPI_File_close(&fh);
```

Consumer

```
/* SPMD MPI-IO write/read coupling*/
double Matrix[dim1][dim2]; /* on each rank */
…
MPI_File_open(MPI_COMM_WORLD, filename,…,&fh);
MPI_File_get_size(fh, &total_number_of_bytes);
my_offset
   = MYRANK*total_number_of_bytes/NUMRANKS;
MPI_File_seek(fh, my_offset, MPI_SEEK_SET);
MPI_File_read(fh,Matrix,dim1*dim2,MPI_DOUBLE, …);
MPI_File_close(&fh);
```

UDJ

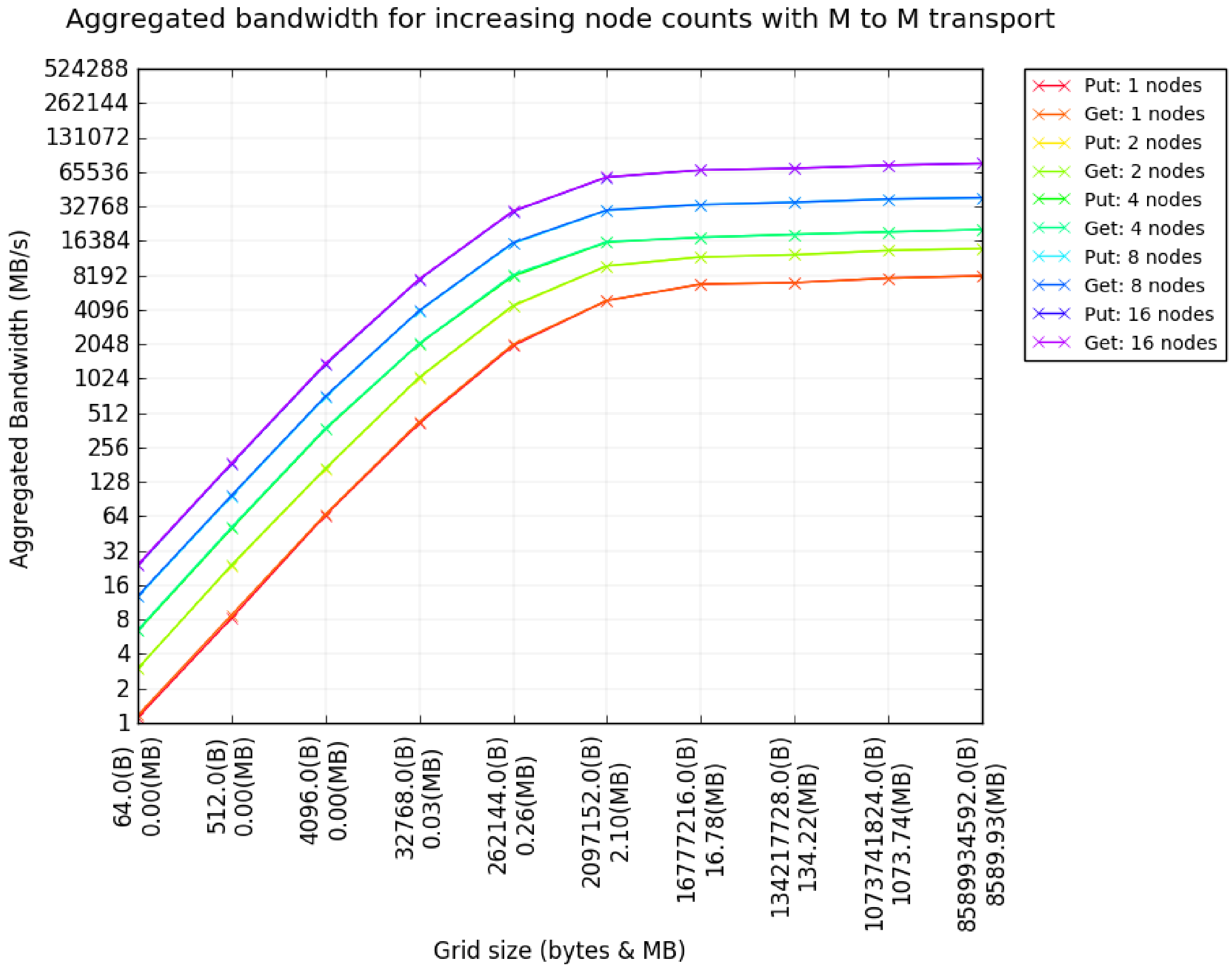
```
/* SPMD write/read coupling*/
double Matrix[dim1][dim2];
…
sender_dist
  =udj_create_dist_cyclic1d(
    numranks,put_ranks,{dim1,dim2});

receiver_dist
  =udj_create_dist_cyclic1d(
    numranks,get_ranks,{dim1}{dim2});

cdo_shape= {dim1,dim2}; /* rank-local size of data */

/* producer: */
udj_put_sync(Matrix,cdo_shape,sizeof(double),
             sender_dist,receiver_dist,cdoid);

/* consumer: */
udj_get_sync(Matrix,cdo_shape,sizeof(double),
             receiver_dist,sender_dist,cdoid);
```

Actual transport method selected at run time:

FS, Datawarp, Dataspaces, RADOS, MPI

Transparent cross-job RDMA network communication (DRC)

UDJ 0.3.2 on MPI-DPM - baseline M:M transfer

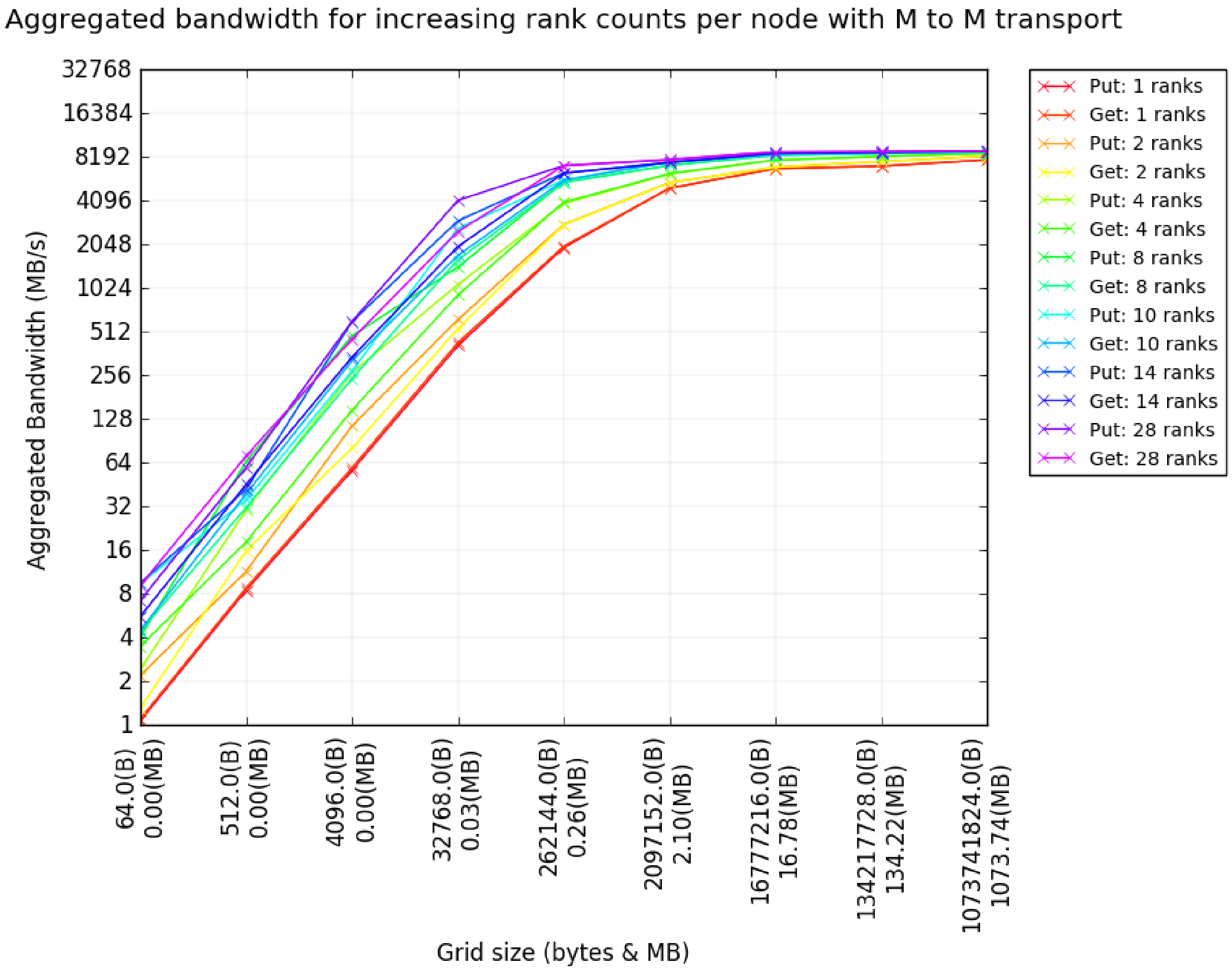


Cray XC30/40 Broadwell 28, 64G+ RAM, Aries interconnect

$(\text{numnodes}*n) \times n \times n$ data sets
1 rank per node

- Block-cyclic distribution that happens to end up requiring 1:1 transfer
- Redistribution to TDOs
- Aggregation of consecutive TDOs
- Chunking (2G default, tunable)

UDJ 0.3.2 on MPI-DPM – on-node scaling M:M transfer



$(\mathrm{numnodes}{*}n) \times n \times n$ data sets
1..28 ranks, 1:1 nodes

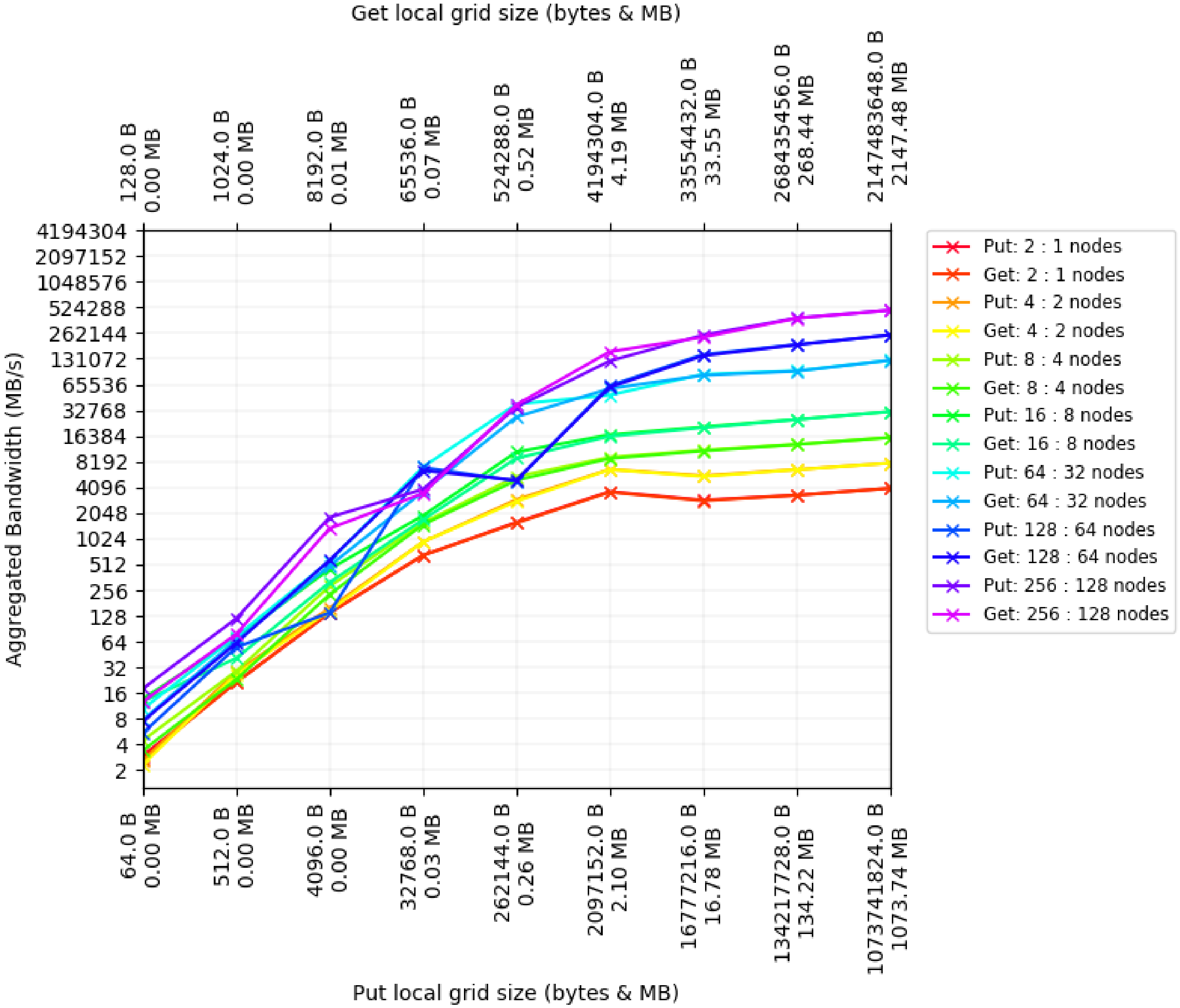
- Block-cyclic distribution that happens to end up requiring 1:1 transfer
- Redistribution to TDOs
- Aggregation of consecutive TDOs
- Chunking (2G default, tunable)

(No dedicated cores or hyperthreads for transport)

Cray XC30/40 Broadwell 28, 64G+ RAM, Aries interconnect

UDJ 0.3.2 on MPI-DPM – ‘easy’ redistribution

Aggregated bandwidth for increasing node counts with 2 to 1 redistribution



Cray XC30/40, 128G+ RAM, Aries interconnect

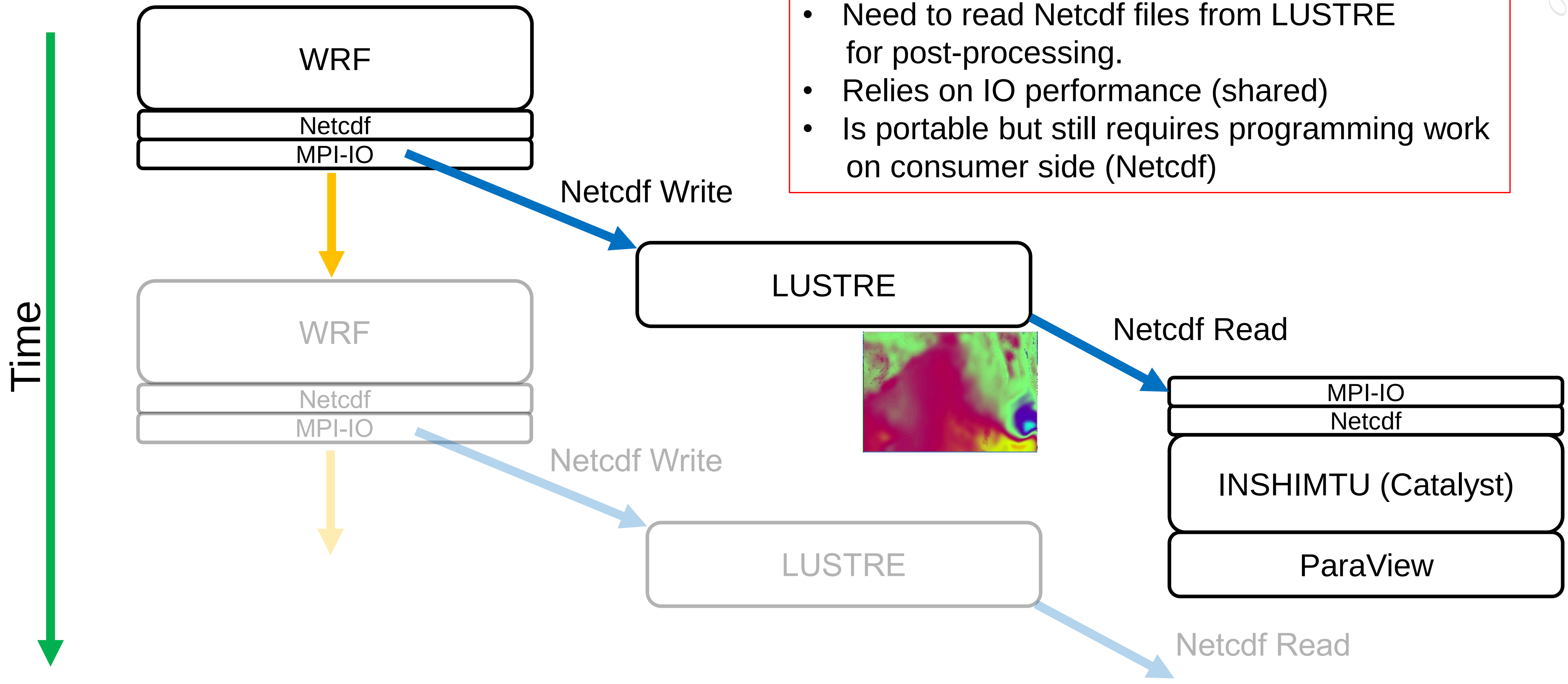
$k \times k \times k$ blocks
2:1 rank ratio, 1 rank per node
Last dimension of receiver grid accommodates process grid change

Aggregation of small (non-consecutive) TDOs (tunable)

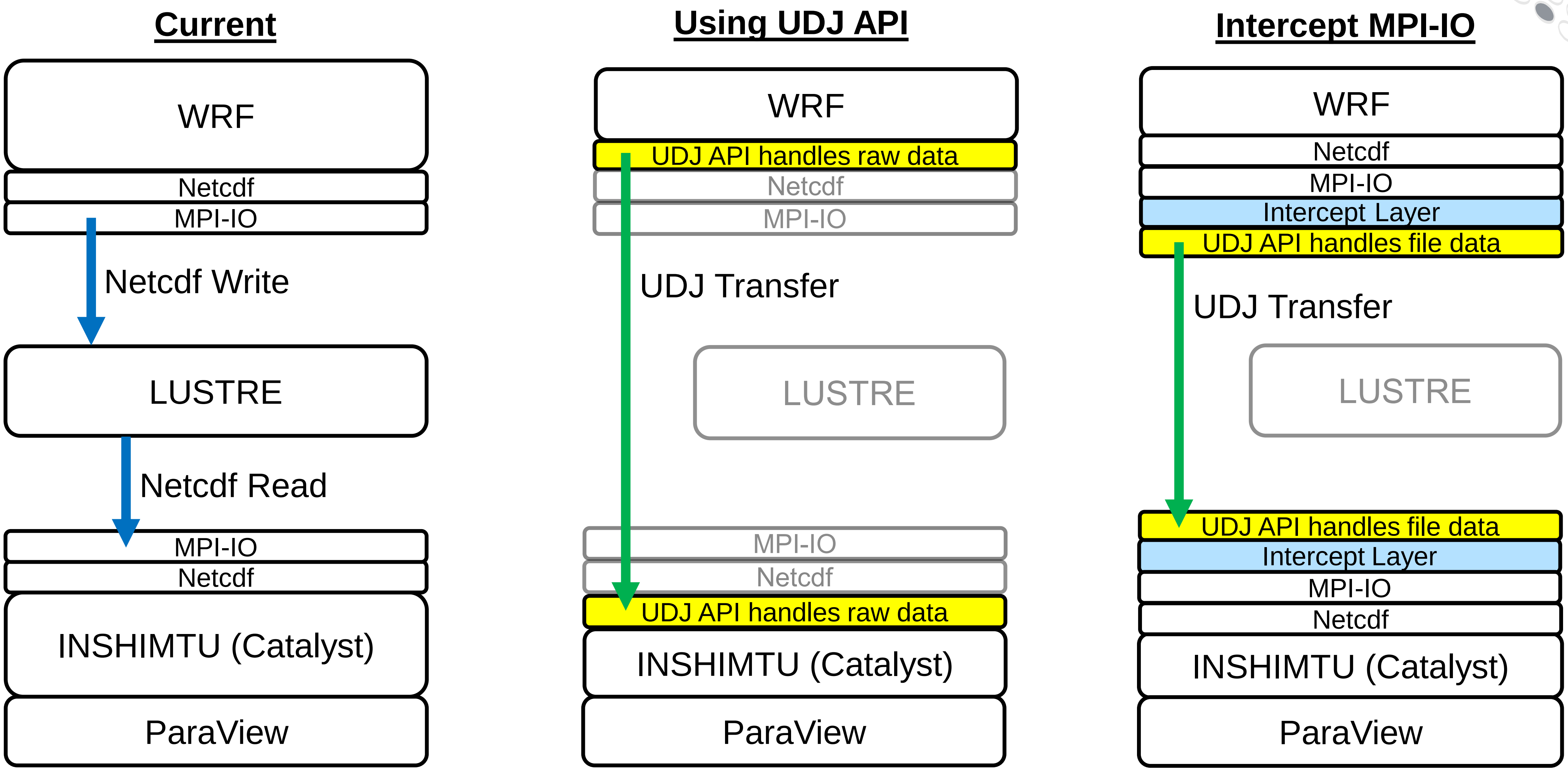
Largest grid yields 3’670’016 TDOs per sender rank

KVL Current Workflow for WRF

- Need to read Netcdf files from LUSTRE for post-processing.
- Relies on IO performance (shared)
- Is portable but still requires programming work on consumer side (Netcdf)

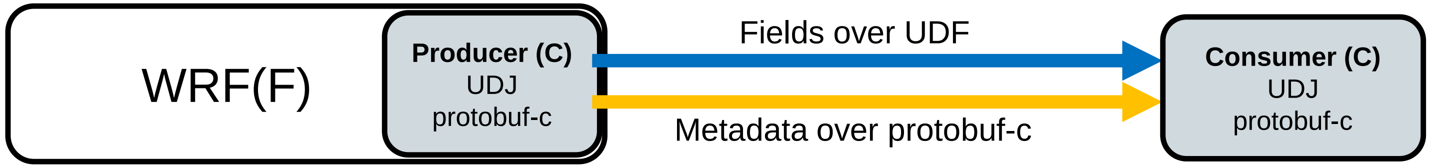


KVL Workflow with UDJ (Options)



Implementation with UDJ-API

- **Intercepted WRF before Netcdf output**
  - Used iso_c_binding to pass fields and metadata to a C routine (producer) which is called by WRF.
  - The producer calls UDJ (put) for the fields and metadata is transferred via protobuf-c
  - Initialization of parallel environment descriptor right after MPI_Init_thread in WRF. An appropriate communicator is passed to WRF.
- **Using dummy consumer written in C**
  - Receives metadata in protobuf-c format and runs UDJ (get)
  - Running consumer and WRF in MPMD mode with SLURM.



Time Comparisons (Simple Example)

- **Between Netcdf output and UDJ transfer to consumer.**
  - Time includes the transfer of metadata.
  - Data for two files (one per domain). Both rather small and written sequentially to LUSTRE.
    - Note that the Netcdf write time is NOT the pure IO time.

File Size [MB]	Netcdf Write [s]	UDJ Transfer [s]	Savings [%]
92	6.28	4.58	**27.14**
78	6.03	4.00	**33.63**

- **Substantial savings for both domains observed**
  - Netcdf data still has to be read by the consumer.
  - Need to compare with distributed IO and larger cases.

Acknowledgements

Co-funded by the Horizon 2020 programme
of the European Union

- HBP Pre-Commercial Procurement : UDJ development
- MAESTRO H2020-FETHPC-2017
  - https://www.maestro-data.eu/
- EPIGRAM-HS H2020-FETHPC-2017
  - https://epigram-hs.eu
- Plan4res EU project : Data Model, mixed transports
  - https://www.plan4res.eu/
- MCSA-ITN EXPERTISE : data redistribution approaches
  - www.msca-expertise.eu/

CRAY®

EMEA RESEARCH LAB