CRAY®

EMEA RESEARCH LAB

Explicit Data Movement Machinery on Cray systems:
Time-to-solution is not the only game in town

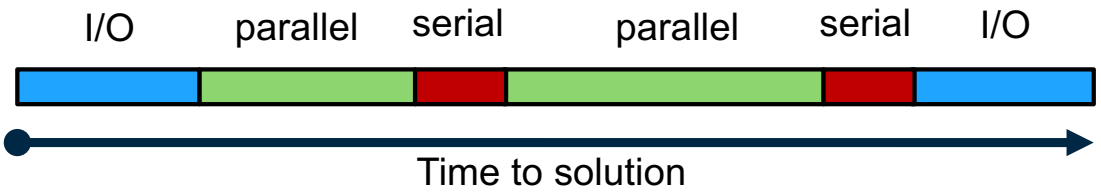
Adrian Tate, Director, CERL

Utz-uwe Haus, Alessandro Rigazzi, Tim Dykes, Edward Fauchon-Jones,
Karthee Sivalingam, Harvey Richardson, Nina Mujkanovic, Aniello Esposito, Clement Foyer

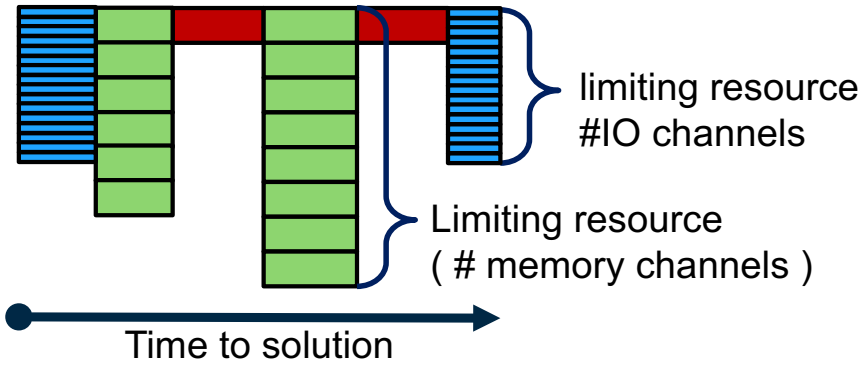
On my travels I hear:

- Data Movement is key
- Massive interest in flexibility and insulation/abstraction
  - Cpu / accelerator
  - Memory
  - Interconnect
  - Abstraction layers to not tie into an architecture
- Technical Collaboration is a must
  - Challenges are too hard!
  - Memory is 20x slower than 1980
  - The complex memory hierarchy isn't even here yet!
    - When it does come, how will we use it?
- HPC + AI : yes but in more than DL frameworks
- Time-to-solution is not the only game in town

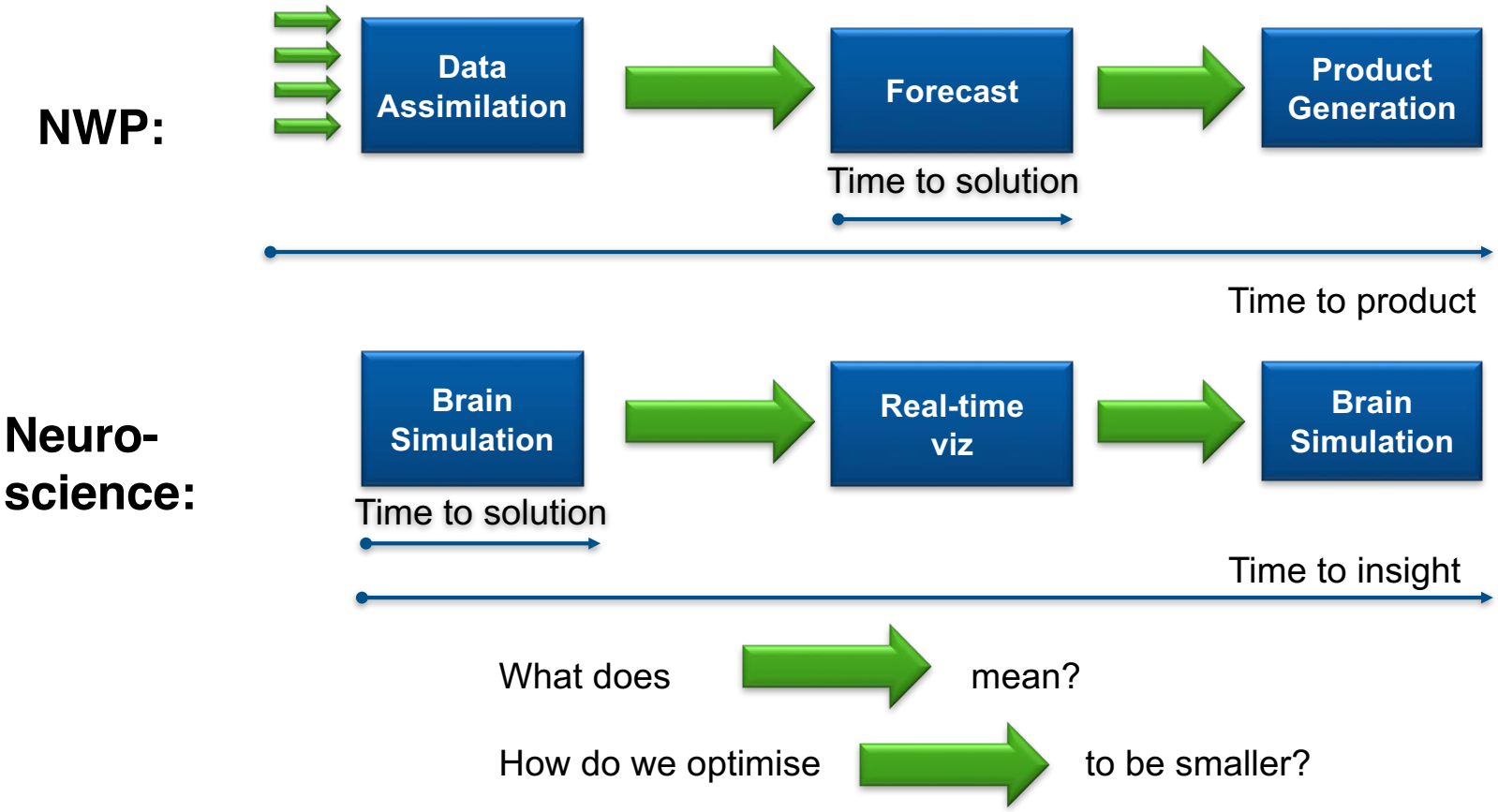
Time to solution

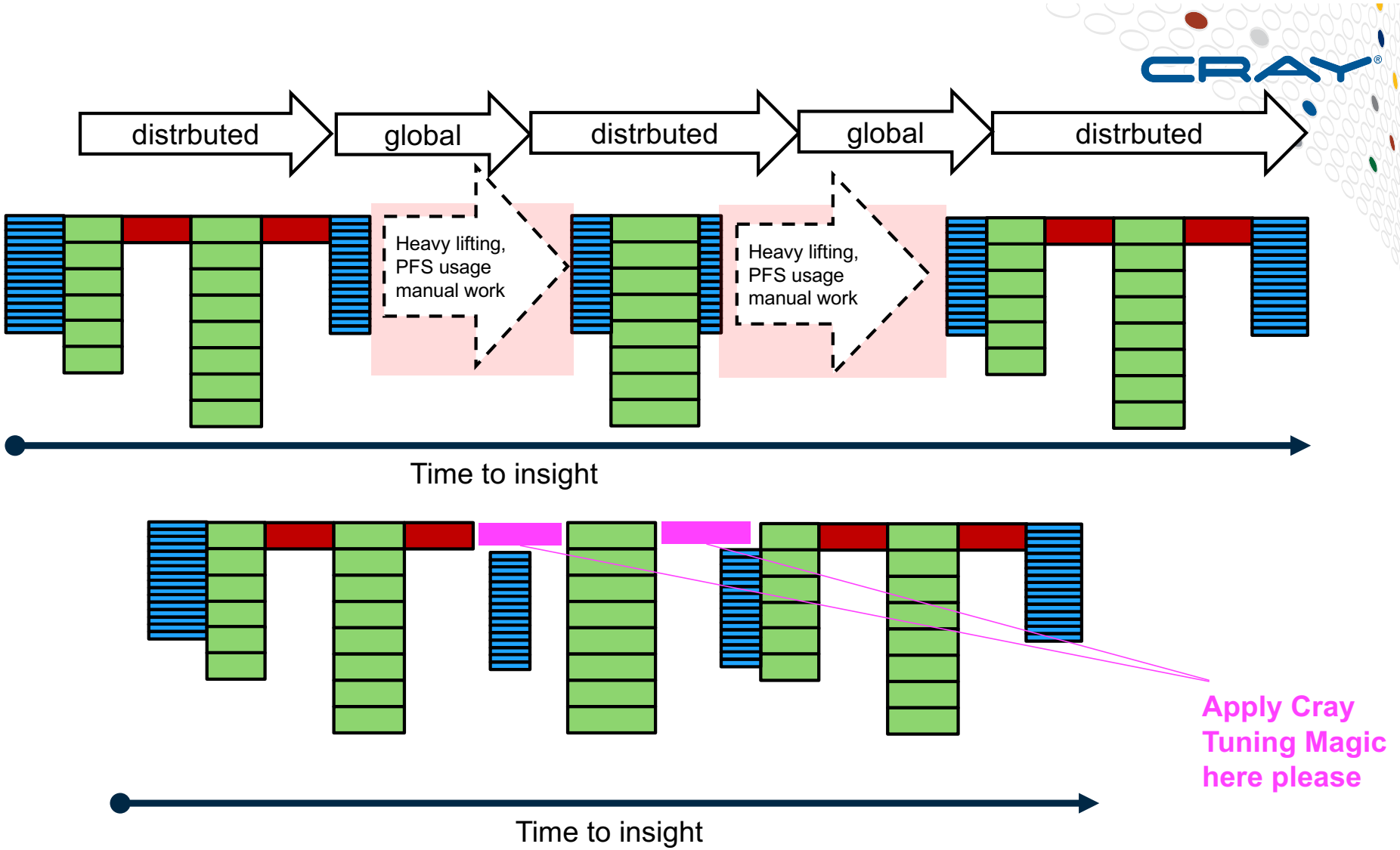


- Many forms of parallelism
- Algorithmic advances
- Code optimization (Compiler & hand)
- ISA features
- Programming Models
- Performance Abstraction
- Systems Software / Operational
- Network, memory increases



Time to scientific product / insight

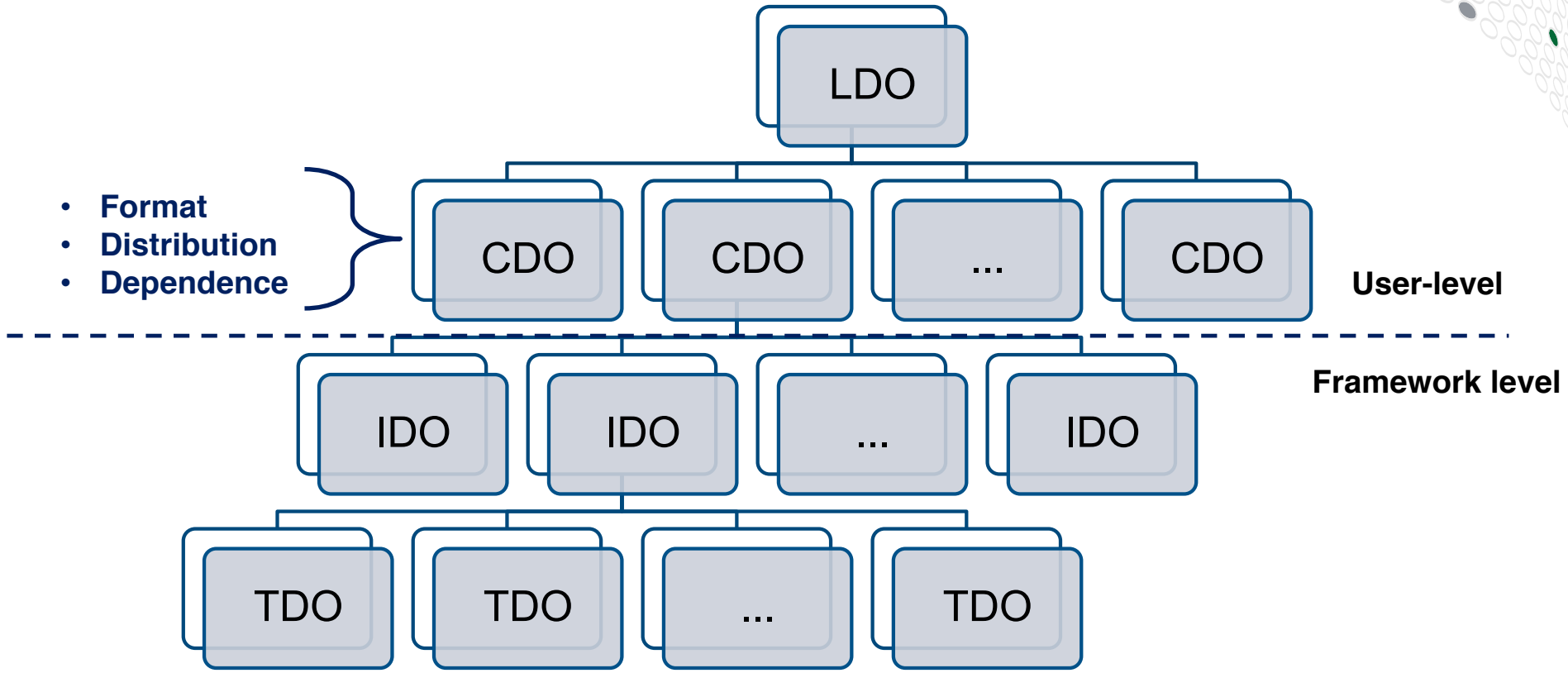



distrbuted
global
distrbuted
global
distrbuted
Heavy lifting,
PFS usage
manual work
Heavy lifting,
PFS usage
manual work
Time to insight
Apply Cray
Tuning Magic
here please
Time to insight

How to start looking at this problem

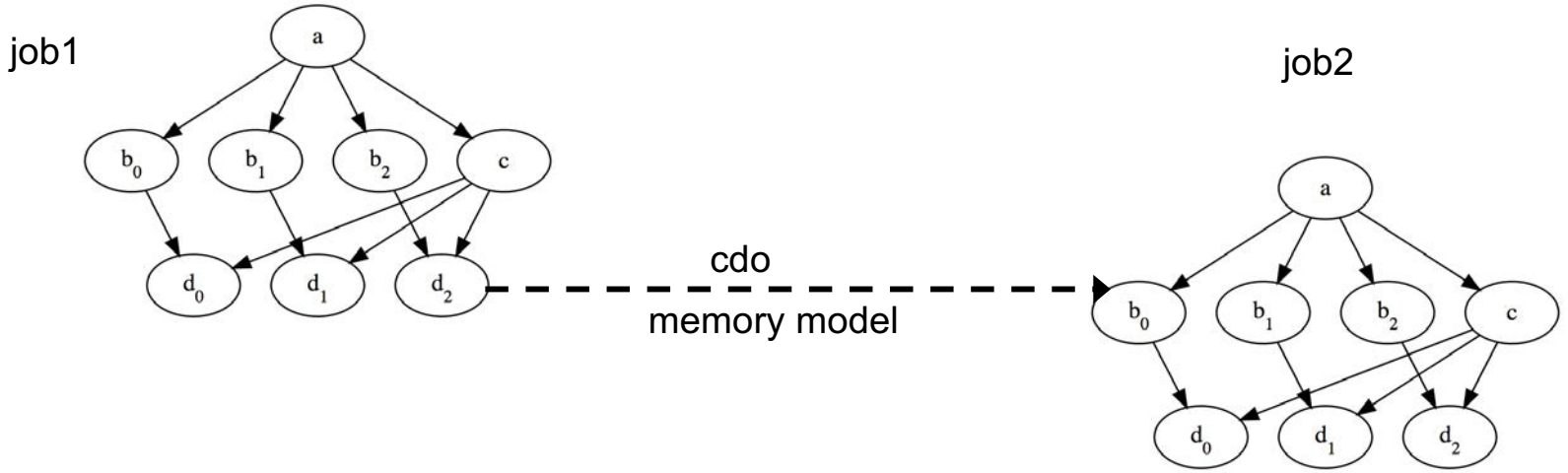
- Lacking in abstractions to tackle the problem
  - No operators to minimise
  - MPI, PGAS express data movement semantics inside job
  - Distinct jobs need to export credentials
  - How to represent the data, its format, its distribution
- Most importantly – parallel data needs to be redistributed
  - Ignoring the fact that data is parallel is ignoring the general problem

Cray Data Object Model



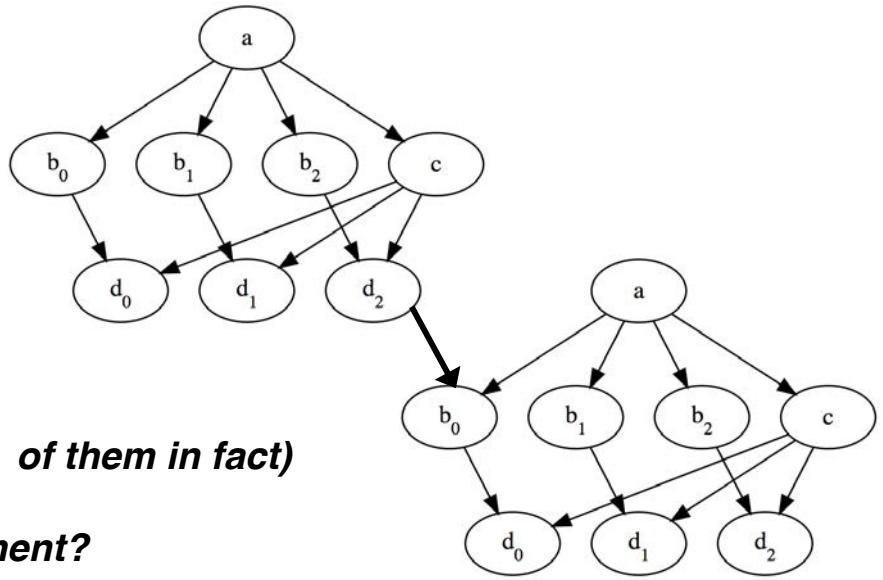
“Octopus” Concept and Project

1. Use object hierarchy to express data dependencies (at a task level)
2. Build a useful model of memory hierarchy
3. Resolve task graph into execution graph
4. Execute graph with data in correct resource



Octopus Concept

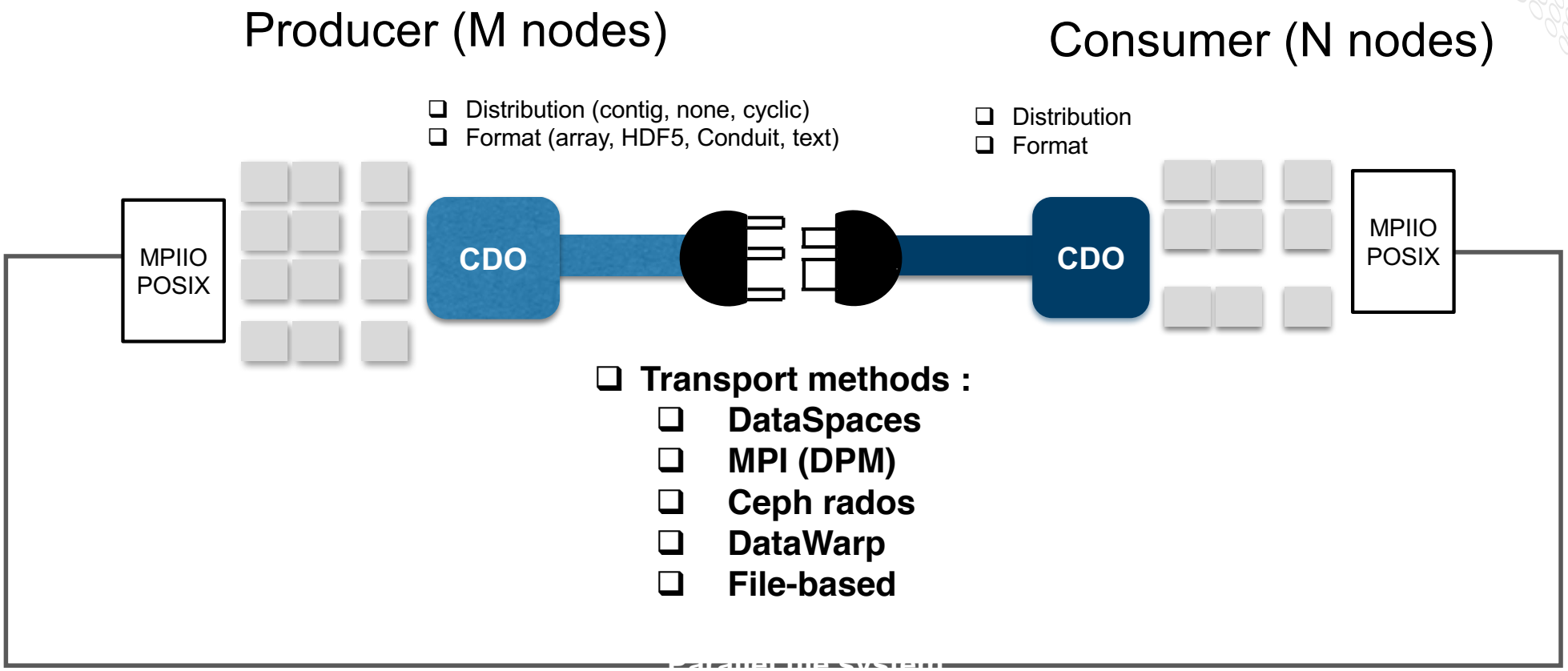
1. Use CDO to represent data dependencies
2. Build a useful model of memory hierarchy
3. Resolve task graph into execution graph
4. Execute graph with data in correct resource



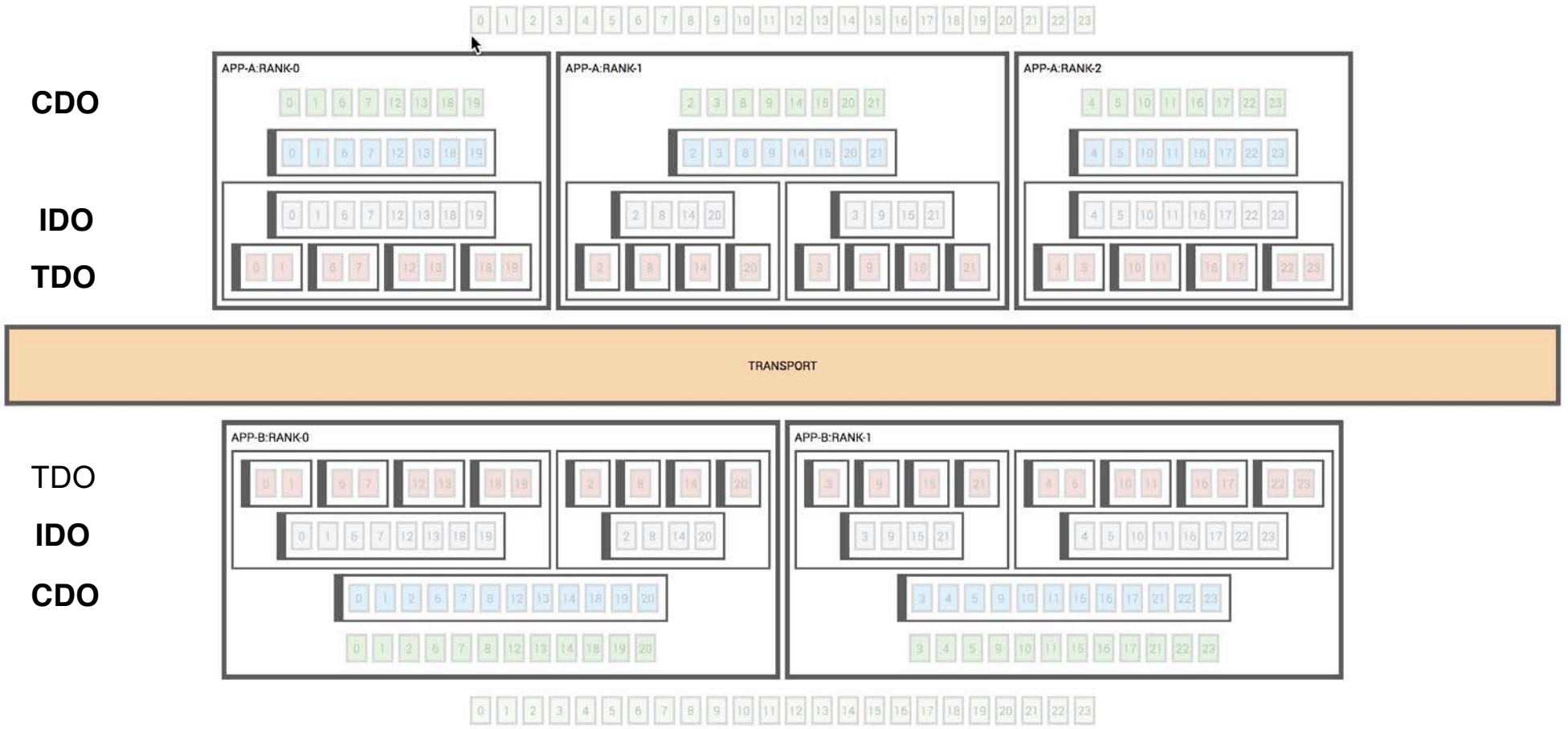
There are no shortage of tools to execute task graph (185 of them in fact)

How many of those can manage the (parallel) data movement?

Universal Data Junction (UDJ)

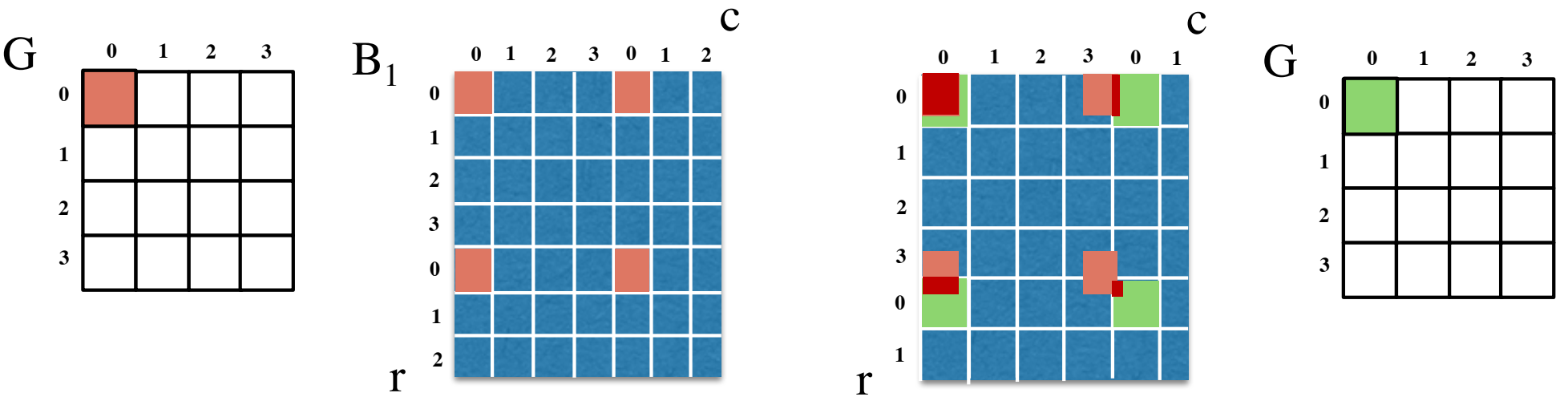


How the redistribution map onto Core Data Object Model



Non-triviality of Producer-Consumer Redistribution

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

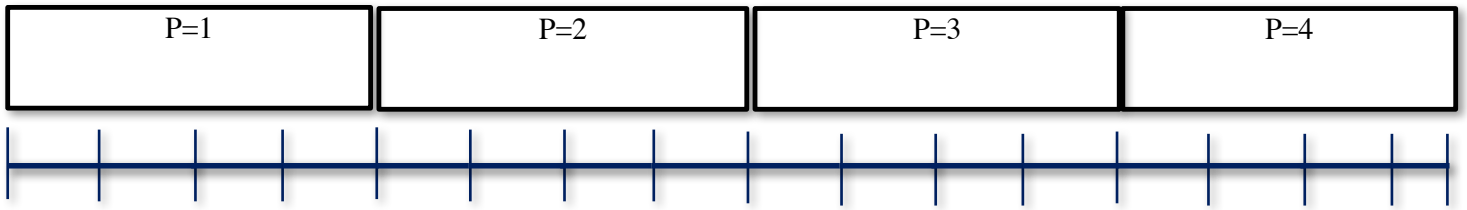


- Re-distributed according to new distribution scheme $D_2=(G, B_2)$ on same grid G
- Must communicate the non-trivial intersection data (red) for every process pair

Classical Redistribution algorithm

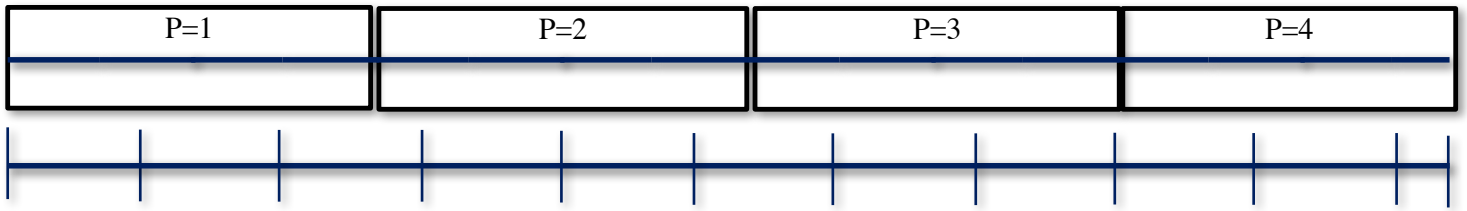
- For each d in #dimensions
- Consider the vector of length r_d
- Divide by blocks of length b_d

Classical Redistribution



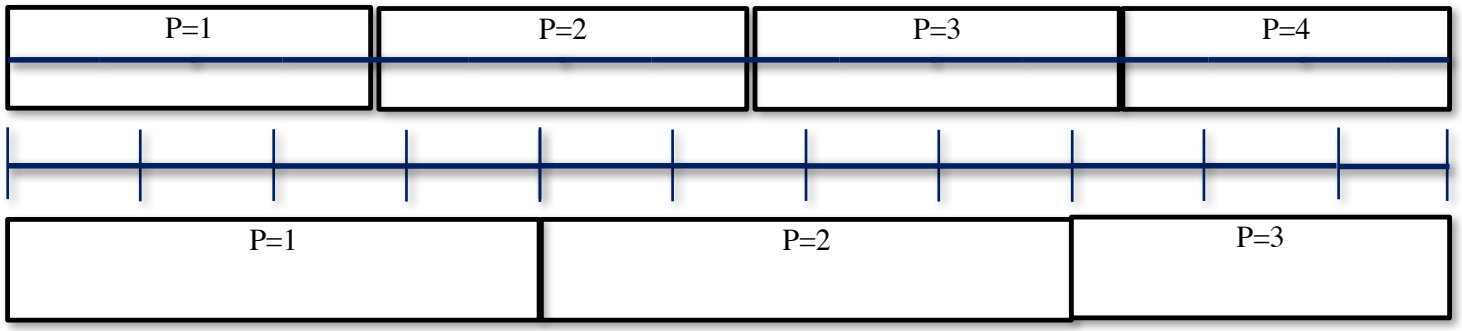
- For each d in #dimensions
- Consider the vector of length r_d
- Divide by blocks of length b_d
- Map the blocks to process rows/column

Classical Redistribution



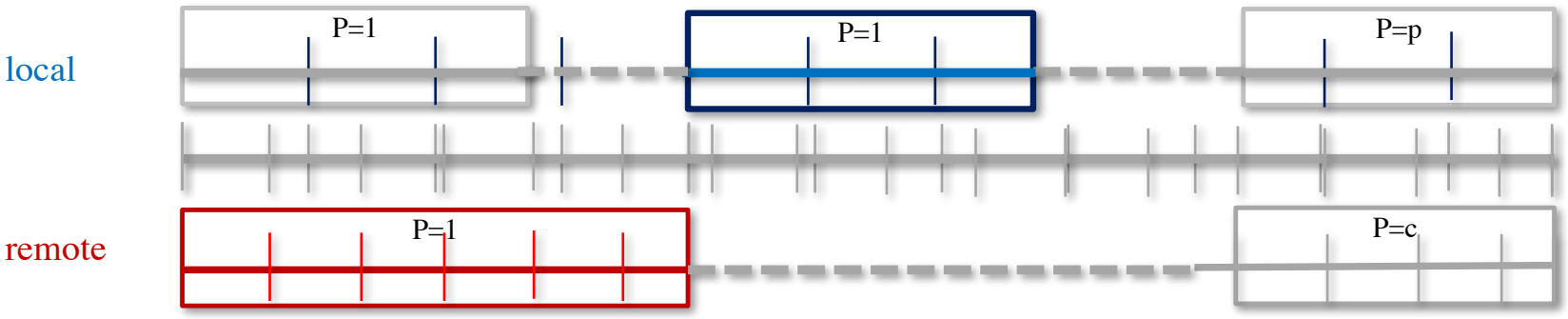
- For each d in #dimensions
- Consider the vector of length r_d
- Divide by blocks of length b^{prod}_d
- Map the blocks to process rows/column
- Divide consumer by blocks of length b^{cons}_d

Classical Redistribution



- For each d in #dimensions
- Consider the vector of length r_d
- Divide by blocks of length b^{prod}_d
- Map the blocks to process rows/column
- Divide consumer by blocks of length b^{cons}_d

Classical Redistribution



On each producer rank:

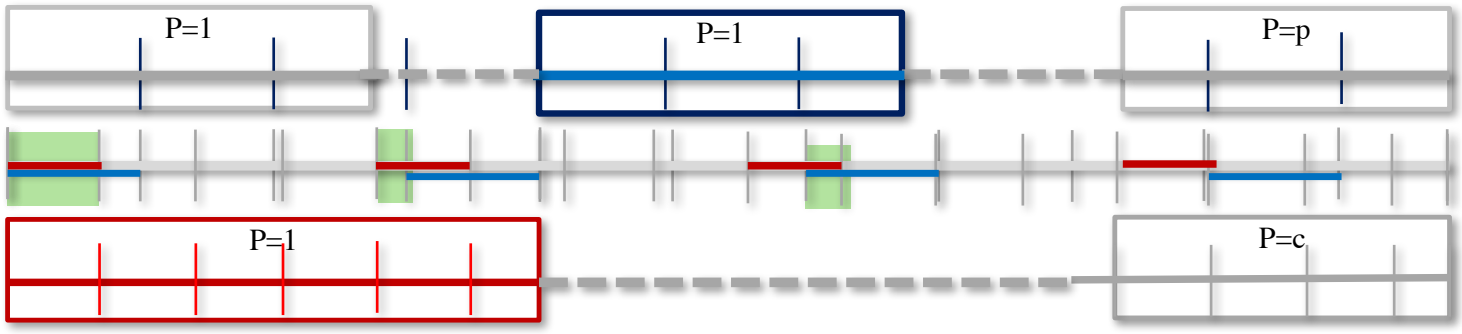
For each d in #Dimensions

For each p in length(consumer_grid(d))

For each loc in #NumLocalBlocks

For each rem in #NumRemoteBlocks

Classical Redistribution



On each producer rank:

For each d in #Dimensions

For each p in length(consumer_grid(d))

For each loc in #NumLocalBlocks

For each rem in #NumRemoteBlocks

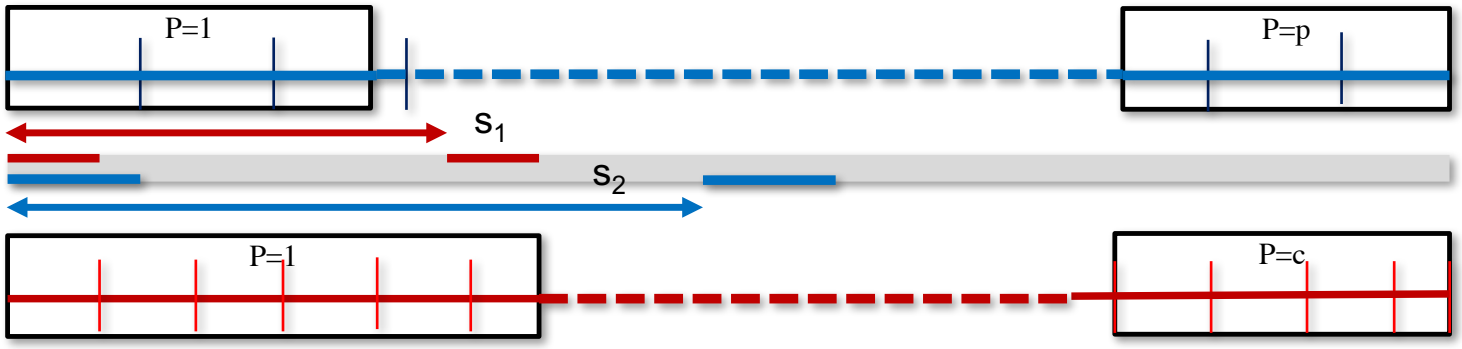
```
if MAX(loc2glob(loc),loc2glob(rem))<
       MIN(loc2glob(loc+b1),loc2glob(rem+b2)) → Add to intersection
```

Intersection = $i_1 \times i_2 \ldots i_d$

Complexity: O(#Dim . P . C . n^{local} . n^{remote})

Ignores three types of periodicity!

Exploit Periodicity of Intersection

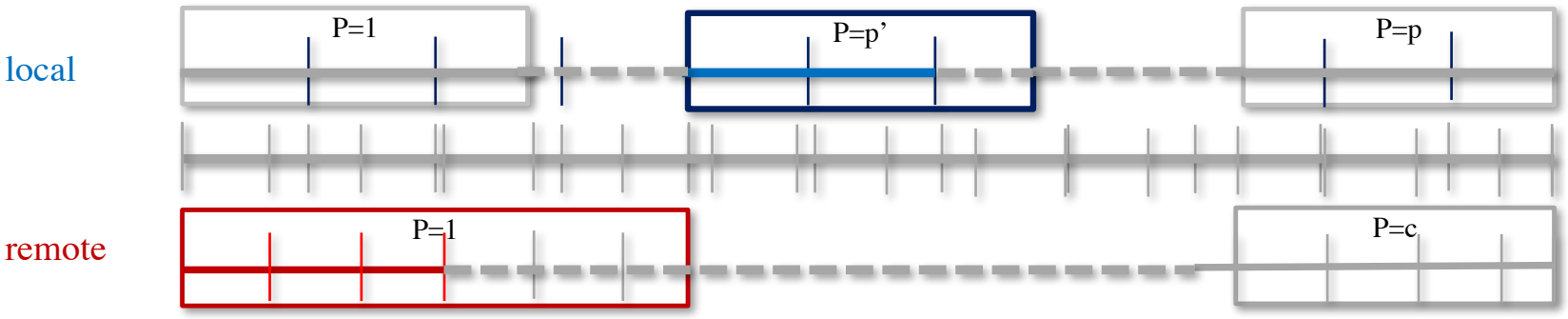


$s = LCM(s_1, s_2)$

Theorem: For a process pair (p_1 , p_2) successive blocks of local data are separated globally by s_{1l} and successive blocks of remote data are separated by s_2 then for every element i in the intersection, element $i + s = LCM(s_1, s_2)$ will also be in the intersection

Source: Guo/Nakata *"A Framework for Efficient Data Redistribution on Distributed Memory Multicomputers"*

Guo-Nakata / FALLS Redistribution



On each producer rank:

For each d in #Dimensions

For each p in length(consumer_grid(d))

For each loc in #NumLocalBlocks

For each rem in #NumRemoteBlocks

Seems unnecessary

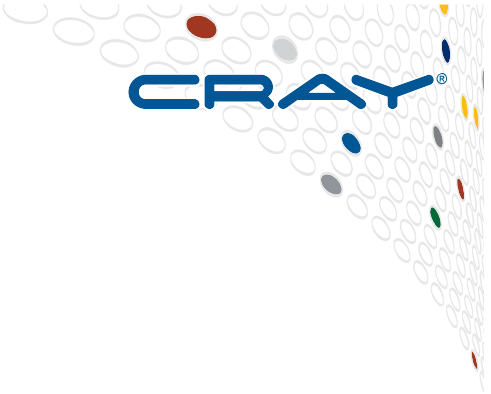
```
if MAX(loc2glob(loc),loc2glob(rem))<
        MIN(loc2glob(loc+b1),loc2glob(rem+b2)) → Add to intersection
```

Intersection = $i_1 \times i_2 \ldots i_d$

$$Complexity : O(\#Dim \,.\, P \,.\, C \,.\hat{n}_{local}\, \hat{n}_{remote})$$

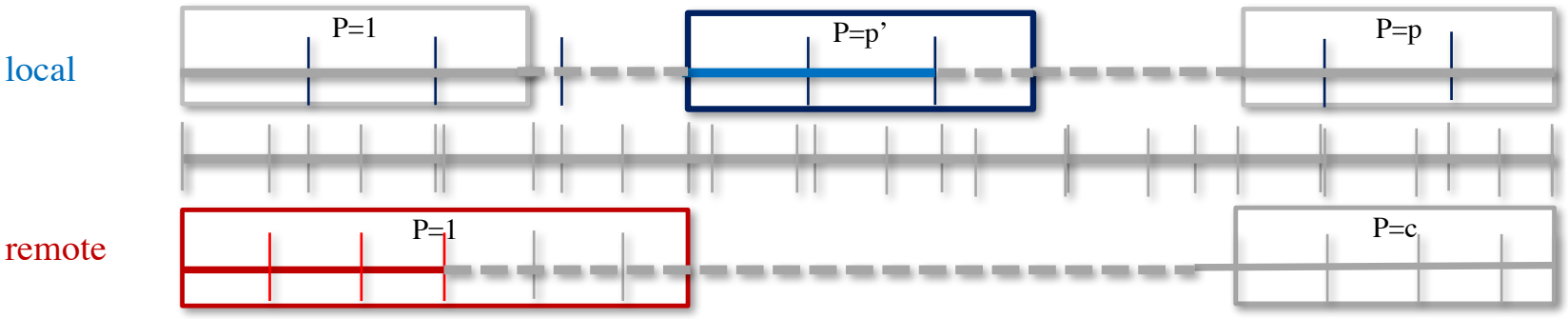
Still ignores 2 types of periodicity!

Exploit Periodicity of Remote Data



- Remote data is of a periodic form
- We can remove one loop if we construct periodic (modular) relations

Exploit Periodic Remote Data



On each producer rank:

For each d in #Dimensions

For each p in length(consumer_grid(d))

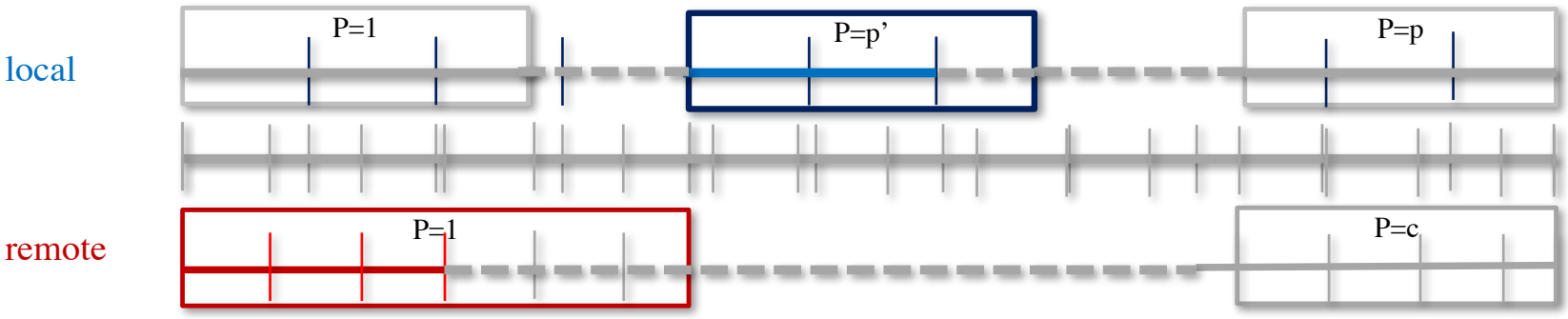
For each loc in #NumLocalBlocks

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

Complexity: O (#Dim . P . C .$\hat{n}_{local}$)

Intersection = i_1 x i_2 … i_d

Exploit Periodic Remote Data



On each producer rank:

For each d in #Dimensions

For each p in length(consumer_grid(d))

Loop order prevents further optimisation

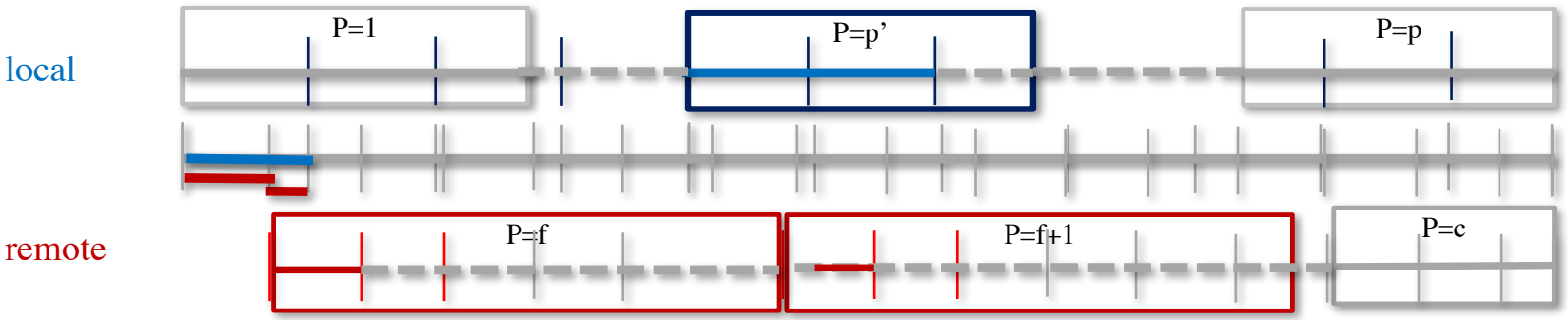
For each loc in #NumLocalBlocks

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

Intersection = $i_1 \times i_2 \ldots i_d$

$$\underline{\textit{Complexity: } O(\#Dim \,.\, P \,.\, C \;.\hat{n}_{local})}$$

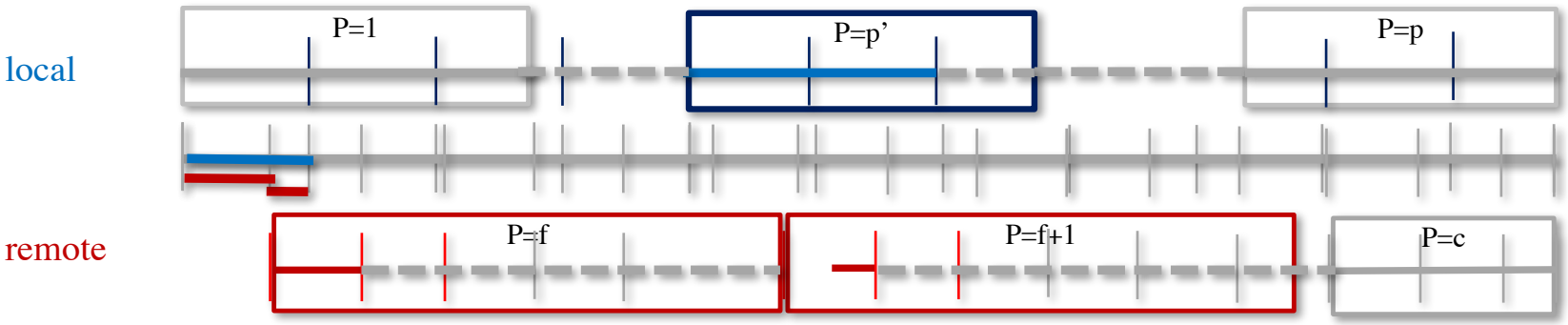
Exploit Periodic Nature of Grid Data



***Theorem**: If local block **l** shares intersection with remote processor **f**, then if an adjacent block of length b2 also exists, that block will share intersection with remote processor **(f+1)% c**, where c is the length of the consumer grid*

Source: Foyer and Tate, "Efficient data redistribution for Producer-Consumer Grids" (xarchiv)

Exploit Periodic Nature of Grid Data



On each producer rank:

For each d in #Dimensions

For each p in length(consumer_grid(d))

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

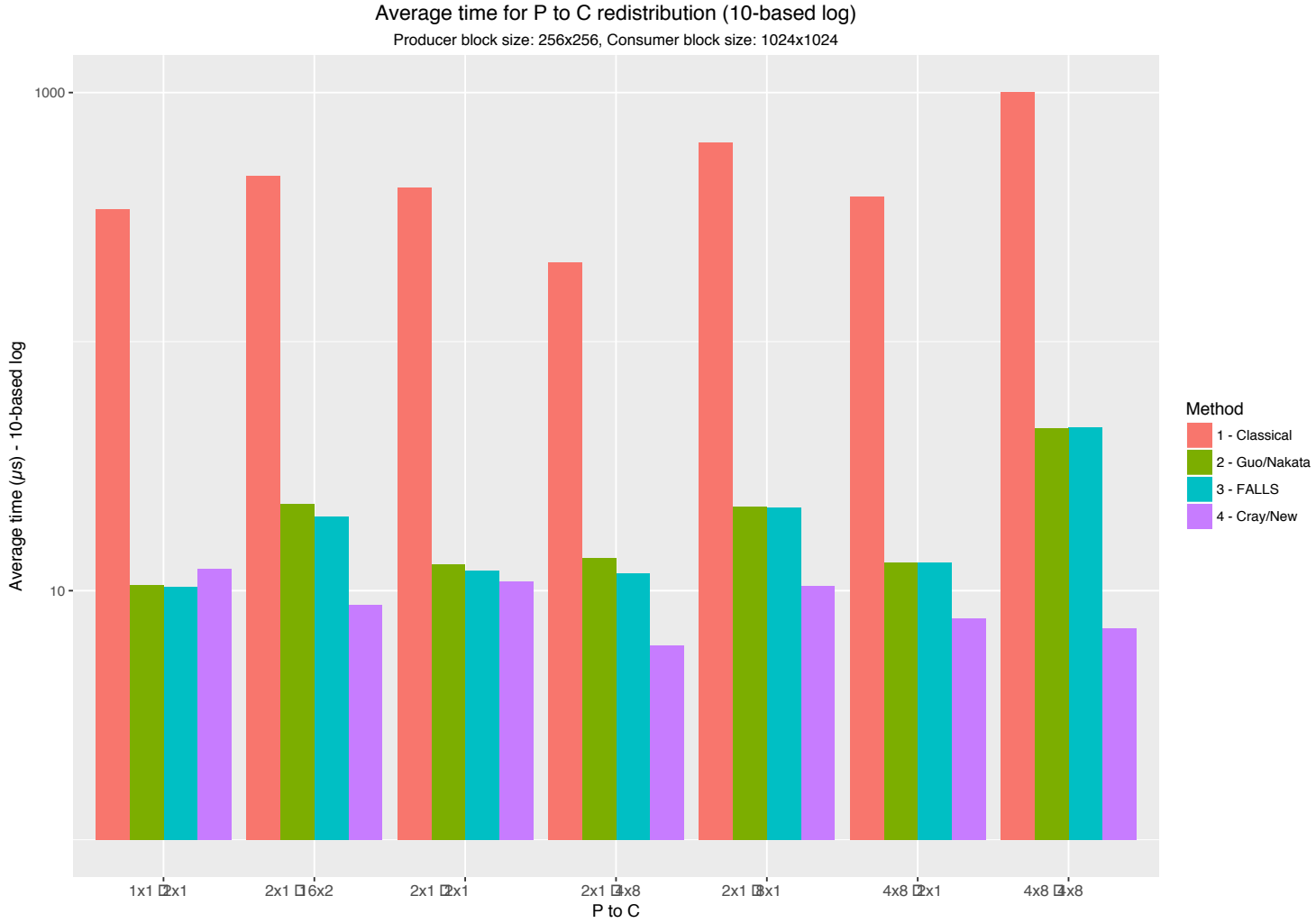
For each sub in b1 / b2

```
→ Add sub to intersection
```

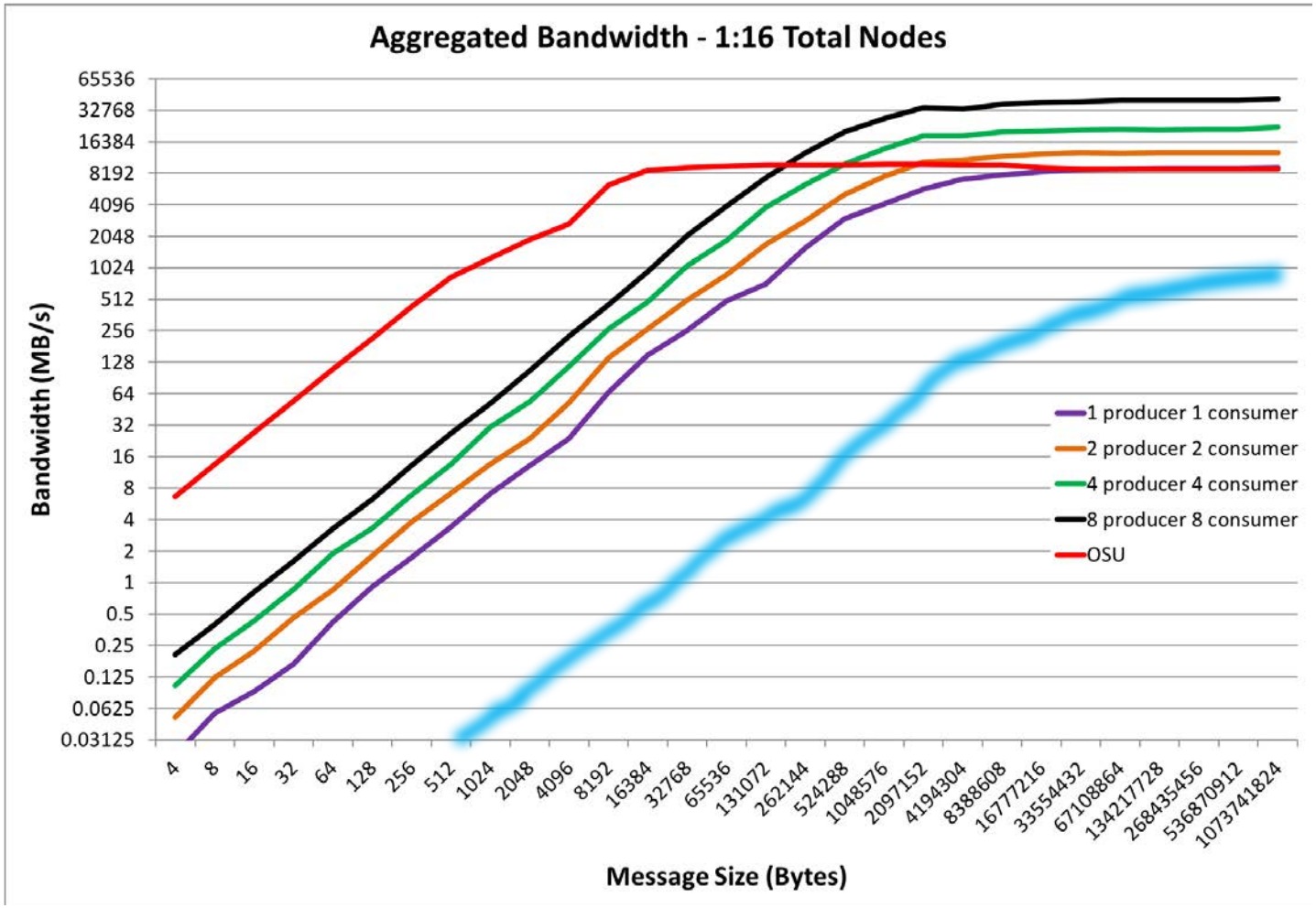
Intersection = $i_1 \times i_2 \ldots i_d$

Complexity: O(#Dim . P . $\hat{n}_{local}$ *. b1/b2)*

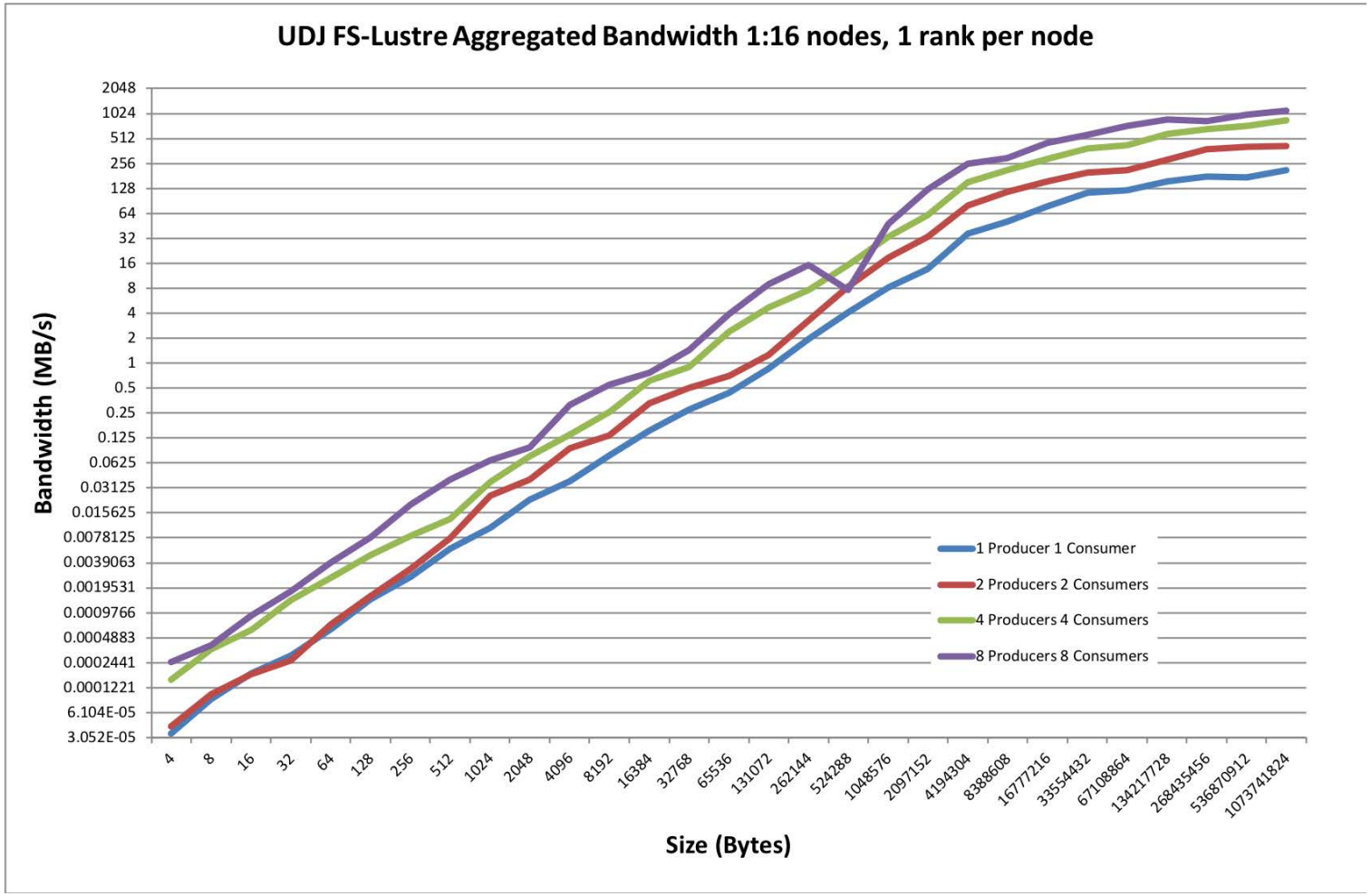
Results – M:N Node Redistribution



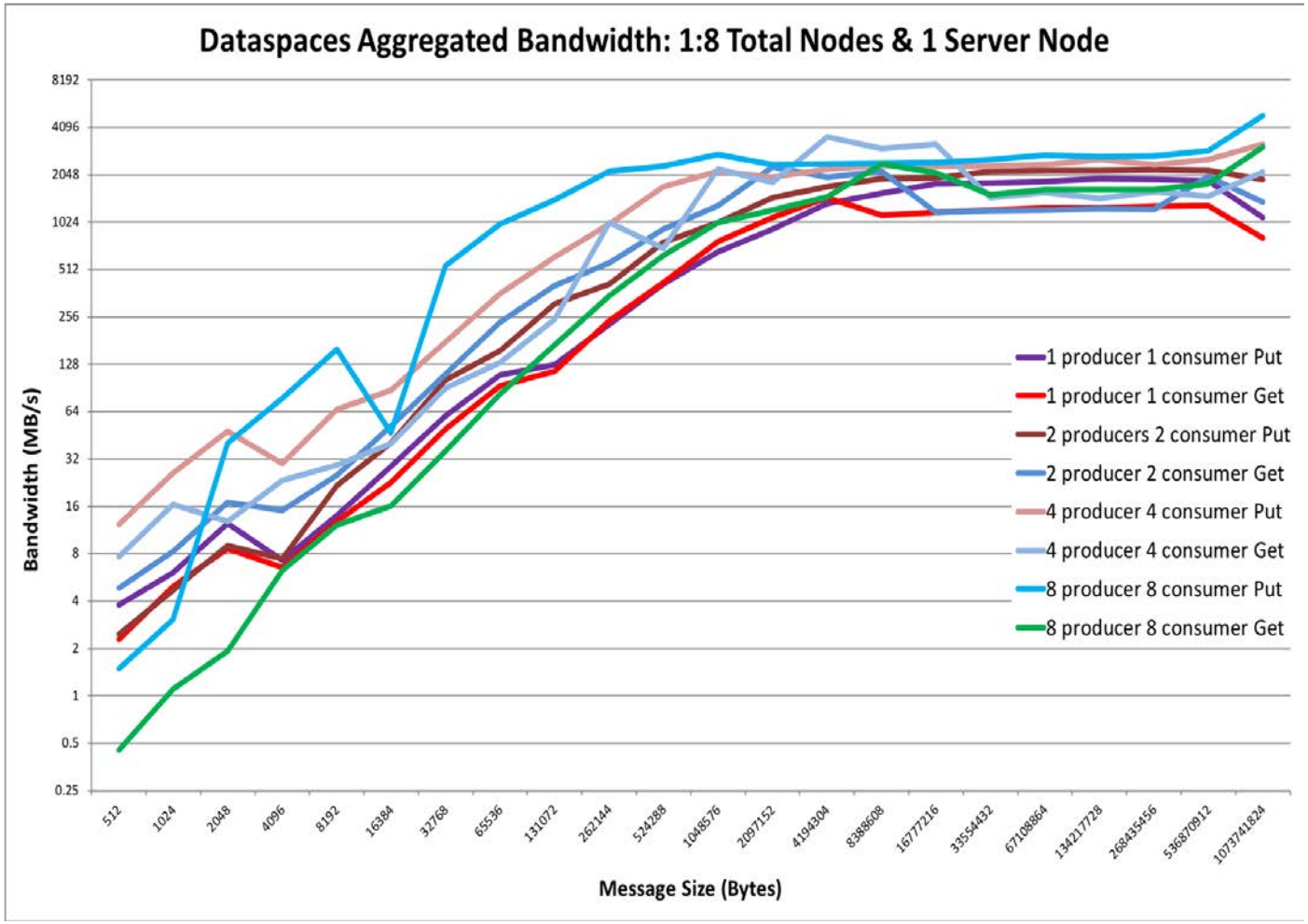
Results UDJ - MPI



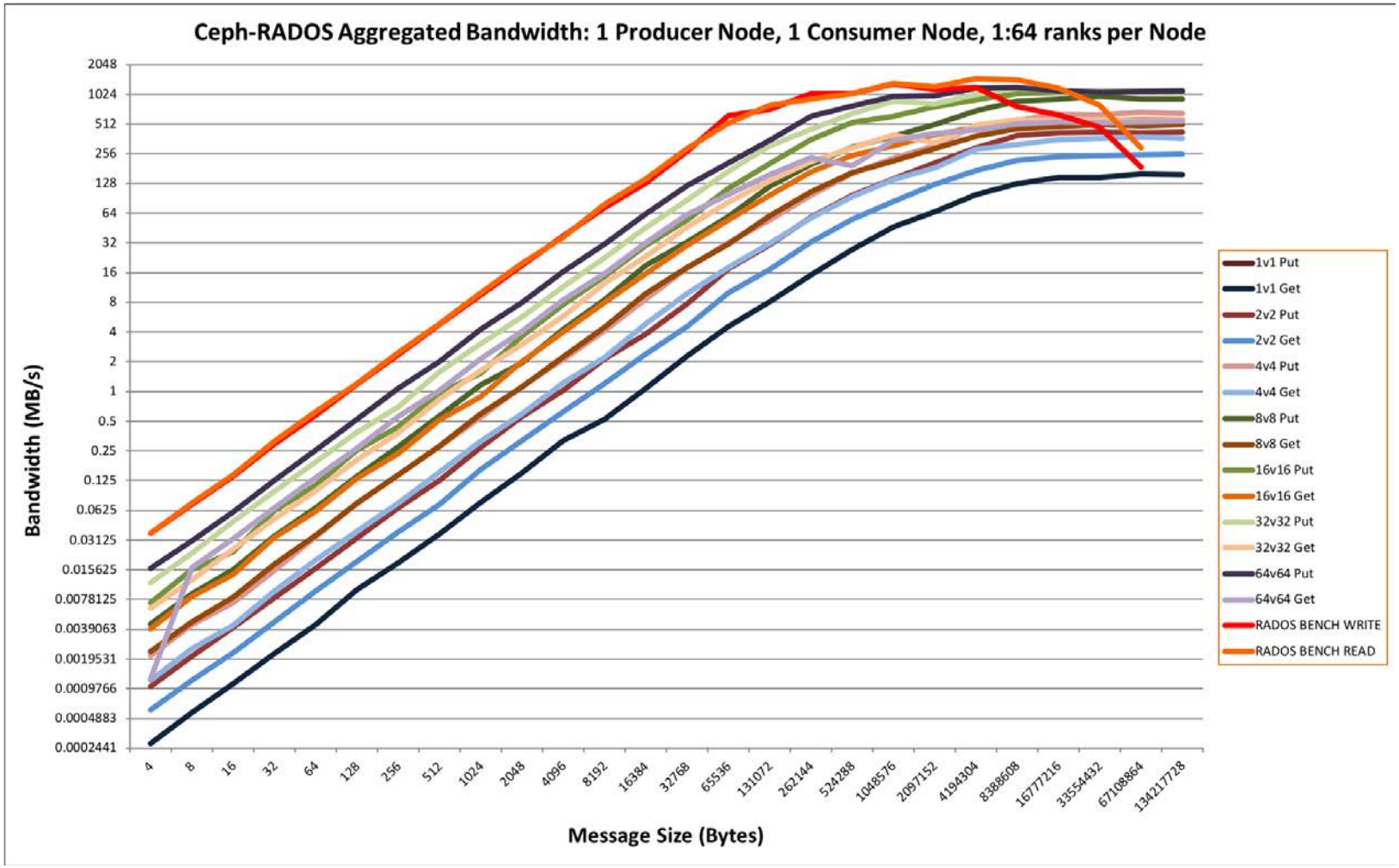
Results UDJ - Lustre



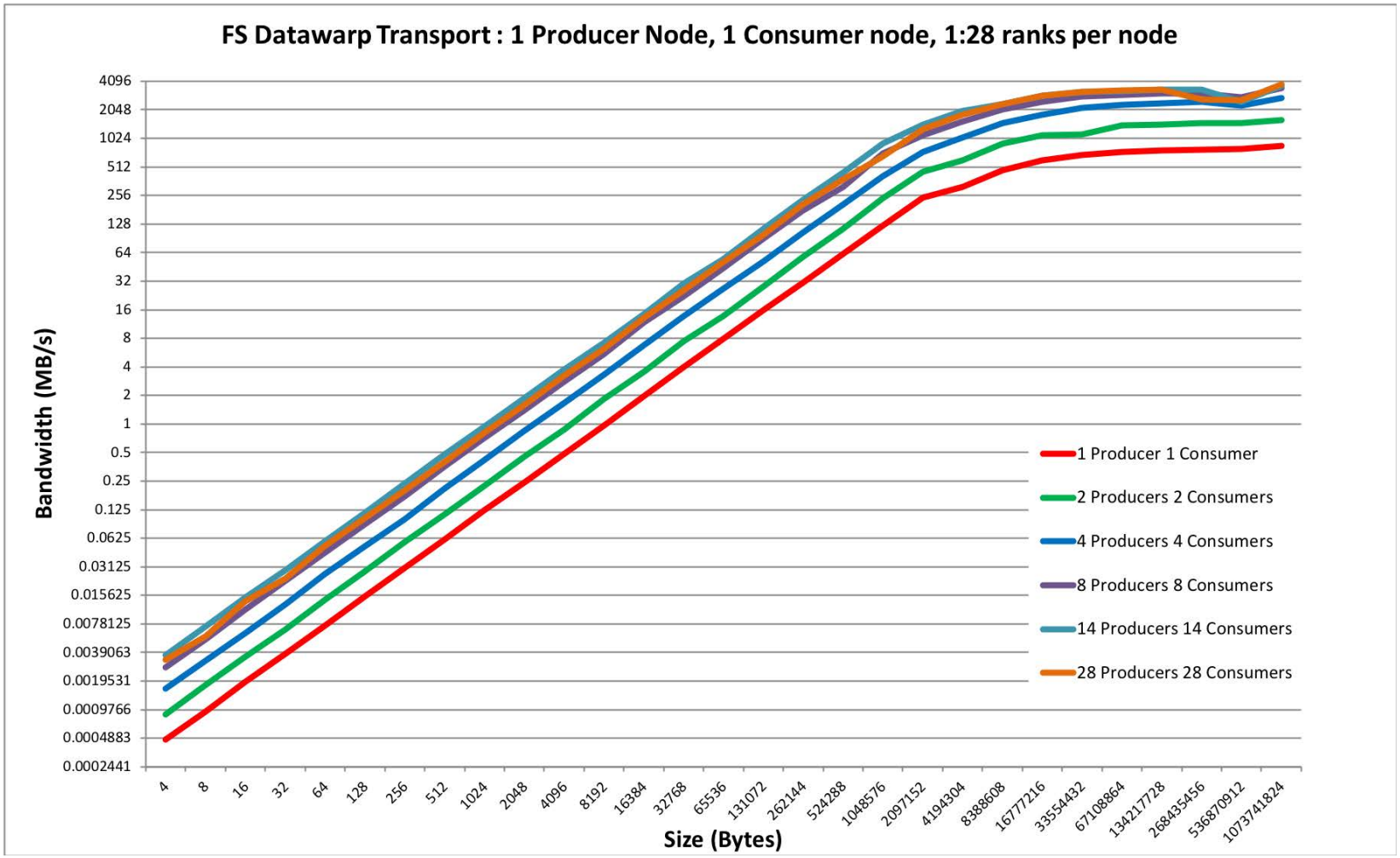
Results – UDJ DataSpaces



Results UDJ – Ceph RADOS



UDJ on DataWarp



Where Next?

- Memory-and data-Aware workflow middleware

- Memory Hierarchy Abstraction Programming

- Asynchronous Task models with data migration

KAUST Cray Centre of Excellence

- Implicit usage of NVRAM 

- Redistribution programming abstraction

- Customised workflow management on Shaheen?

Acknowledgement and Call

- Special thanks to CERL team and collaborators for UDJ
- Funding bodies
  - HBP-PCP : UDJ development

Human Brain Project

  - Plan4res EU project : Data Model
    - https://www.plan4res.eu/

  - MCSA-ITN EXPERTISE : data redistribution approaches
    - www.msca-expertise.eu/

  - Centre for Doctoral Training in Data-Intensive Science (U. Cardiff)

- Contribute to UDJ and Octopus (BSD license)
  - Contact **adrian@cray.com** for access pre-release