

Supplementary material

Transitioning from file-based HPC workflows to streaming data pipelines with openPMD and ADIOS2

Franz Poeschel^{1,4}, Juncheng E⁵, William F. Godoy³, Norbert Podhorszki³, Scott Klasky³, Greg Eisenhauer⁶, Philip E. Davis⁷, Lipeng Wan³, Ana Gainaru³, Junmin Gu², Fabian Koller⁴, René Widera⁴, Michael Bussmann^{1,4}, and Axel Huebl^{2,4}

¹ Center for Advanced Systems Understanding (CASUS), D-02826 Görlitz, Germany

² Lawrence Berkeley National Laboratory (LBNL), Berkeley 94720, California, USA

³ Oak Ridge National Laboratory (ORNL), Oak Ridge 37830, Tennessee, USA

⁴ Helmholtz-Zentrum Dresden-Rossendorf (HZDR), D-01328 Dresden, Germany

⁵ European XFEL GmbH (EU XFEL), D-22869 Schenefeld, Germany

⁶ Georgia Institute of Technology (Georgia Tech), Atlanta 30332, Georgia, USA

⁷ Rutgers University (Rutgers), New Brunswick 08901, New Jersey, USA

An integrated workflow for flexibly efficient data distribution

Codes traditionally often pursue a random-access pattern at loading data from the filesystem, mainly due to the simplicity of it. We propose an alternative workflow where the chunk distribution from data source to sink is determined by distribution algorithms, taking into consideration properties of data provenance for optimization purposes. A chunk distribution strategy's performance depends on the involved applications, the compute hardware and the specification of a compute job. A data processing pipeline where two communicating applications share the same compute nodes has different requirements from such a pipeline where both applications run on different system partitions.

As a result, this paper proposes a workflow that makes the choice of a chunk distribution algorithm part of the dynamically specified IO configuration, allowing users to quickly adapt to changing circumstances. Some analyses may require algorithms with domain-specific knowledge on logical constraints to uphold certain *read constraints*.

As a note, data-distribution aware processing (data provenance and data paths), while highly relevant in streaming setups, has also been shown to improve file-based parallel processing [2].

The workflow for two cooperating applications to share massive simulation data in an algorithmically distributed manner is sketched in figure 1 on the following page. It begins by producing data on the writer side. ADIOS2 stages the written data in the writer side's memory and, upon closing the current step of writing data, informs the reading side about the data available for loading.

Any further step on the writer's side after this point will happen asynchronously, allowing the simulation to proceed and hide IO times in the background. Within the context of openPMD, this meta data defines the structure of fields and species present in the series. Along with the meta data, the writer also sends information on its own topology which the chunk distribution algorithms may take into consideration.

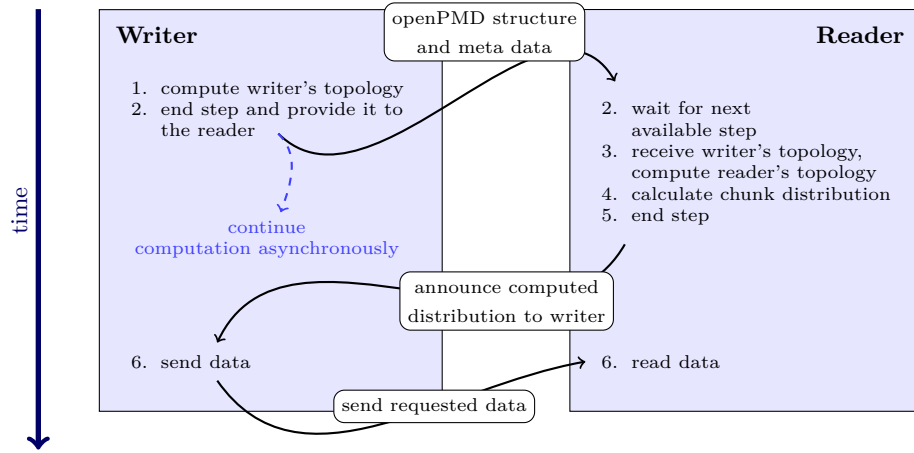


Fig. 1: Calculation of chunks within the workflow of a coupled simulation

Next up is the computation of the chunk distribution on the reading end. Since patterns for data loading might be subject to application-specific constraints, this computation must happen on the reading end within the context of this generic workflow. By receiving the reader's topology with the meta data, computing its own topology and knowing about data available for loading along with data provenance, the reader has all information to make an educated choice on an efficient chunk distribution. We present the details of this step in the following subsection.

The reader announces this distribution to the writer by ending the current step.¹

¹ ADIOS2 theoretically allows an arbitrary number of data loading operations within one step. For performance reasons, this should be avoided, preferring the workflow presented above.

Chunk Distribution Algorithms

Algorithms 1 to 4 on pages 5–6 show in detail how the distribution strategies explained in the paper proceed.

Algorithm 3 adapts a basic Binpacking approximation (Next-Fit from [1]) and it is not immediately obvious why it is correct. We prove that the presented algorithm will actually assign all chunks in the central for-loop. Assume for contradiction a chunk of size x has not found place in any of the $2N$ bins. At least one slot must exist among the $2N$ slots that is filled strictly below half its capacity. Otherwise, the total fill status would be $\geq 2N \cdot \left(\frac{1}{2} \cdot \frac{ext}{N}\right) = ext$, i.e. every data item would have been distributed.

- If $x \leq \frac{1}{2} \cdot \frac{ext}{N}$, the chunk would fit in this slot.
- If $x \geq \frac{1}{2} \cdot \frac{ext}{N}$, the greedy nature of the algorithm would have preferred to put the chunk of size x into the slot over the smaller chunks that are occupying it.

Both situations cannot occur and the correctness of the algorithm is hence proven by contradiction.

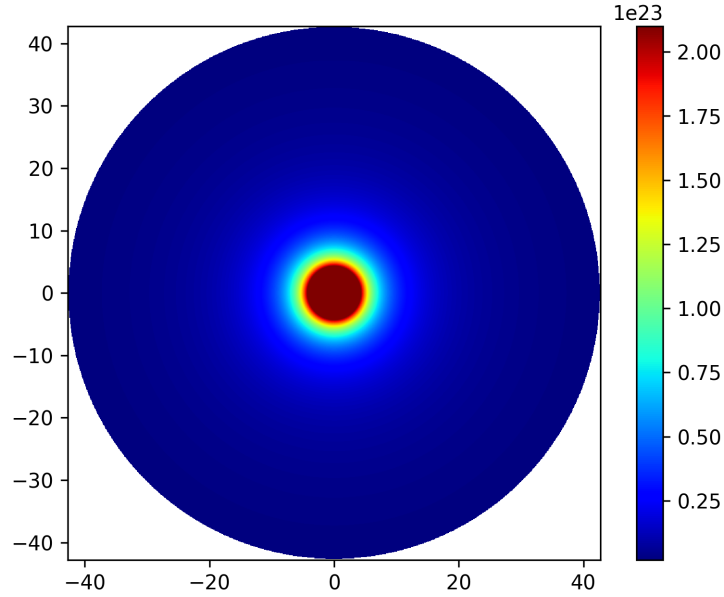


Fig. 2

GAPD: Results

The small-angle X-ray scattering plot for the electrons in a 1536-GPU Kelvin-Helmoltz PIC simulation computed by a loosely coupled GAPD simulation is found in figure 2 on the previous page. The used X-ray energy/wavelength and detection geometry settings are found in the configuration file `in.GAPD`.

Used software versions

Self-built:

- PIconGPU:
<https://github.com/franzpoeschel/picongpu/tree/smc2021-paper>
- GAPD: closed source software, Git tag `smc2021-paper` in private repository
- openPMD-api:
<https://github.com/franzpoeschel/openPMD-api/tree/smc2021-paper>
- ADIOS2: <https://github.com/ornladios/ADIOS2>
 Git hash `bf25ad59b8b15b9f48ddabad65a41f2050d3bd7f`
- libfabric: version 1.6.3a1

Summit modules:

- 1) `gcc/8.1.1`
- 2) `spectrum-mpi/10.3.1.2-20200121`
- 3) `cmake/3.18.2`
- 4) `git/2.20.1`
- 5) `cuda/10.1.243`
- 6) `boost/1.66.0`
- 7) `zlib/1.2.11`
- 8) `libpng/1.6.34`
- 9) `freetype/2.9.1`
- 10) `python/3.7.0-anaconda3-5.3.0`

References

- [1] David Johnson. “Near-Optimal Bin Packing Algorithms”. PhD thesis. Massachusetts Institute of Technology, 1973.
- [2] Lipeng Wan et al. “Improving I/O Performance for Exascale Applications through Online Data Layout Reorganization”. under review. 2021.

Algorithm 1 Round Robin approach

```

1: function DISTRIBUTECHUNKS(partialAssignment, destinationRanks)
2:   let  $N \leftarrow \text{sizeof}(\text{destinationRanks})$ 
3:   let  $\text{currentRank} \leftarrow 1$ 
4:   for  $\text{chunk} \leftarrow \text{unassigned chunks}$  do
5:     Assign chunk to rank  $\text{destinationRanks}[i]$ 
6:      $i \leftarrow i + 1$ 
7:     if  $i > N$  then
8:        $i \leftarrow 1$ 
9:     end if
10:  end for
11: end function

```

Algorithm 2 Slicing the dataset into n -dimensional hyperslabs

```

1: function DISTRIBUTECHUNKS(partialAssignment, destinationRanks, datasetExtent)
2:   Split the dataset into hyperslabs such that
3:     each destination rank is assigned exactly one cube
4:   let  $\text{cube} \leftarrow \text{current rank's assigned cube}$ 
5:   for  $\text{chunk} \leftarrow \text{leftover chunks}$  do
6:      $\text{chunk} \leftarrow \text{intersect } \text{chunk} \text{ with } \text{cube}$ 
7:     if  $\text{chunk}$  is non-zero in all dimensions then
8:       Assign  $\text{chunk}$  to current rank
9:     end if
10:  end for
11: end function

```

Algorithm 3 Binpacking

```

function DISTRIBUTECHUNKS(partialAssignment, destinationRanks)
  let  $N \leftarrow \text{number of } \text{destinationRanks}$ 
  let  $\text{ext} \leftarrow \sum_{\text{unassigned chunks}} (\text{extent of chunk})$ 
  split unassigned chunks into subchunks of size at most  $\frac{\text{ext}}{N}$ 
  let  $L \leftarrow \text{list of those subchunks, sorted from largest to smallest}$ 
  let  $A \leftarrow \text{array of length } 2N, \text{ containing assigned chunks}$ 
   $\triangleright$  each slot can hold a size of  $\frac{\text{ext}}{N}$ 

  for  $i \leftarrow 1, \dots, 2N$  do
    for  $\text{chunk} \leftarrow L$  in decreasing order do
      if slot  $i$  can additionally hold  $\text{chunk}$  then
        Add  $\text{chunk}$  to slot  $A[i]$ 
      else
        continue
      end if
    end for
    Remove assigned chunks from  $L$ , keeping order
  end for
   $\triangleright$  At this point, all chunks are assigned (see proof)
  for  $i \leftarrow 1, \dots, N$  do
    Assign all chunks in  $A[i] \cup A[N + i]$  to rank  $i$ 
  end for
end function

```

Algorithm 4 Hostname-based distribution

```

1: function DISTRIBUTECHUNKS(incomingChunks, metaInformation)
2:   let sourceChunks be a mapping from hostnames
3:     to chunks on that host
4:   let sinkRanks be a mapping from hostnames
5:     to source rank ids running on that host
6:   for hostname  $\leftarrow$  available hostnames do
7:     let chunks  $\leftarrow$  sourceChunks[hostname]
8:     let ranks  $\leftarrow$  sinkRanks[hostname]
9:     if ranks =  $\emptyset$  then                                      $\triangleright$  No sink process on this host
10:      Report chunks as leftover chunks
11:     else
12:       Assign chunks to ranks,
13:       using an adequate interior chunk distribution algorithm
14:     end if
15:   end for
16:   Use any fallback distribution strategy for the leftover chunks
17: end function

```
