SEA Conference 2018

Decoupled MapReduce using MPI One-Sided

Sergio Rivas-Gomez | KTH
Stefano Markidis | KTH
Erwin Laure | KTH
Keeran Brabazon | ARM
Oliver Perks | ARM
Sai Narasimhamurthy | Seagate (UK)

Introduction > The relevance of I/O

Scientific applications running on current Petascale Supercomputers demand high-performance I/O to read their ever-increasing input sets and store detailed results. Some of these applications include:

HPC Applications

- Space Weather

- Neuroscience

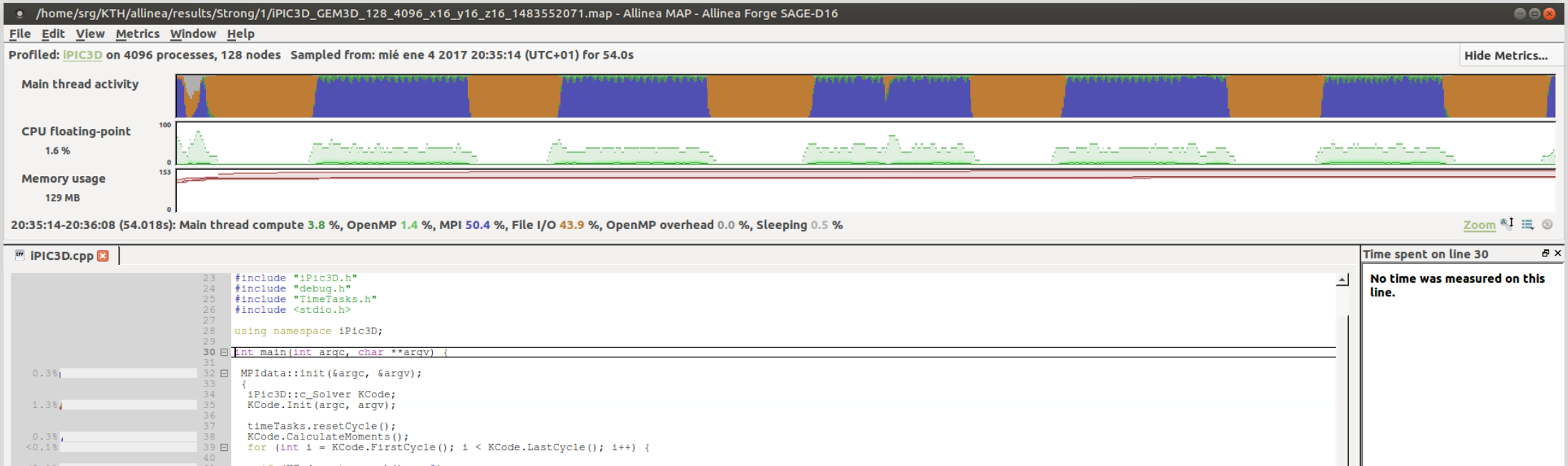
- Visualization

- Others

Image Source: (1) ESA (2) biology.nd.edu (3) TACC Stallion (4) FEniCS-HPC

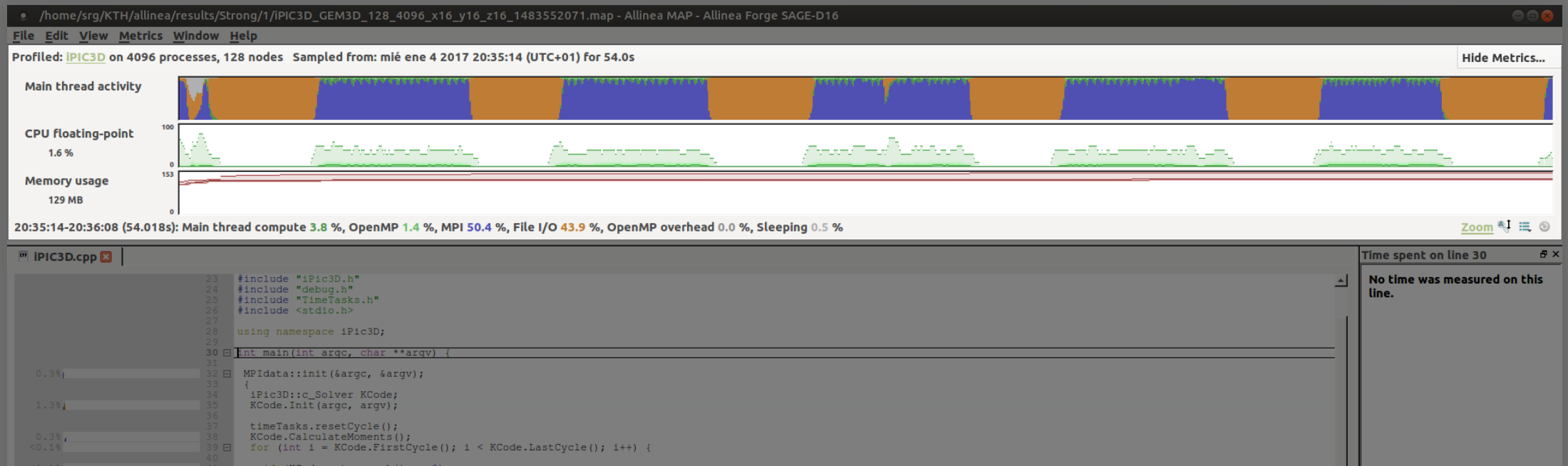
Introduction > Science demands high-performance I/O

Scientists are aggressively taking advantage of the large available computing power to run larger scale and high-fidelity simulations, requiring the movement of large files to / from storage. For instance, most of the scientific applications **use storage devices for pre-/post-processing, and fault tolerance:**



Introduction > Science demands high-performance I/O

Scientists are aggressively taking advantage of the large available computing power to run larger scale and high-fidelity simulations, requiring the movement of large files to / from storage. For instance, most of the scientific applications **use storage devices for pre-/post-processing, and fault tolerance:**



Introduction > Increasing use-cases on HPC

In addition to traditional scientific applications, emerging graph analytics and machine learning applications are becoming common on supercomputers, motivated by the advances in deep learning and convolutional networks:

Big Data & Machine Learning
are trending on HPC

The irregular nature of data analytics pose a large stress on the I/O subsystem, mainly because of the high number of small I/O transactions. In addition, **some of these use-cases require out-of-core execution as well**.

Introduction > Moving away from separate HPC for I/O

Historically, HPC systems have been commonly designed as a large system for computing, and a separate large system that offers persistency support (e.g., through a Distributed File System):

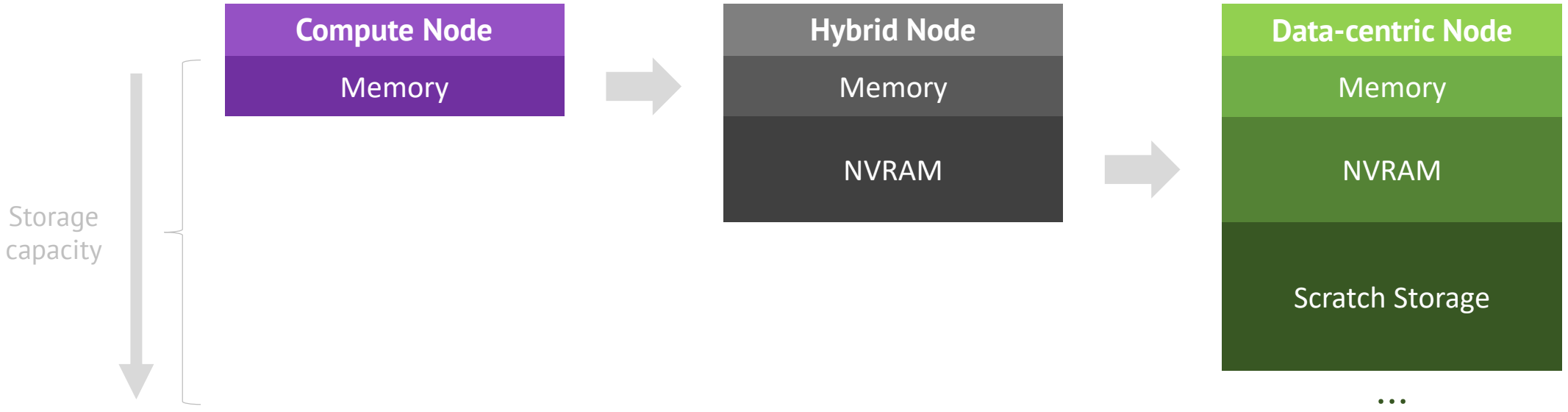
Traditional HPC Systems

Specialized HPC System

The trend for large-scale computer design is diverging from the traditional compute-only node approach, to **hybrid solutions where the data is moved next to the computation**.

Introduction > Increasing heterogeneity per node

Emerging storage technologies are evolving so rapidly that the existing gap between main memory and I/O subsystem performances is thinning. Next-generation supercomputers will feature a variety of Non-Volatile RAM (NVRAM), with different performance characteristics, next to traditional DRAM:



Introduction > Challenges of data-centric trends

Now that data-intensive workloads have become an integral part of large-scale scientific computing, **several challenges arise that must be considered on programming models of upcoming HPC clusters**:

Complexity	Performance	Scalability
Applications becoming extremely complex	High-Performance cannot be guaranteed	Future clusters to feature a concurrency of 100-1000×

Introduction > Challenges of data-centric trends

Now that data-intensive workloads have become an integral part of large-scale scientific computing, **several challenges arise that must be considered on programming models of upcoming HPC clusters**:

Complexity	Performance	Scalability
Applications becoming extremely complex	High-Performance cannot be guaranteed	Future clusters to feature a concurrency of 100-1000×

Introduction > Challenges of data-centric trends

Now that data-intensive workloads have become an integral part of large-scale scientific computing, **several challenges arise that must be considered on programming models of upcoming HPC clusters**:

Complexity	Performance	Scalability
Applications becoming extremely complex	High-Performance cannot be guaranteed	Future clusters to feature a concurrency of 100-1000×

Introduction > Challenges of data-centric trends

Now that data-intensive workloads have become an integral part of large-scale scientific computing, **several challenges arise that must be considered on programming models of upcoming HPC clusters**:

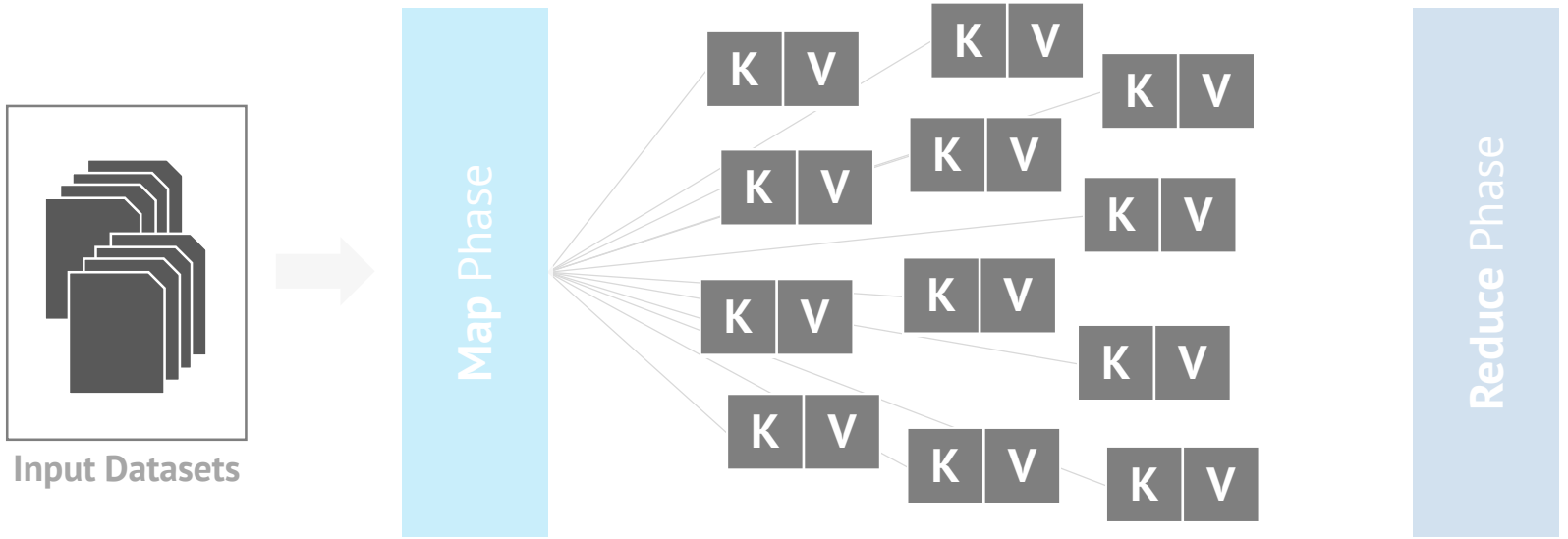
Complexity	Performance	Scalability
Applications becoming extremely complex	High-Performance cannot be guaranteed	Future clusters to feature a concurrency of 100-1000×

Introduction > MapReduce is trending

MapReduce has become one of the preferred programming models to hide the complexity of process and data parallelism. The power of this paradigm resides on the definition of simple `Map()` and `Reduce()` functions, that become highly-parallel operations using complex inter-processor communication:

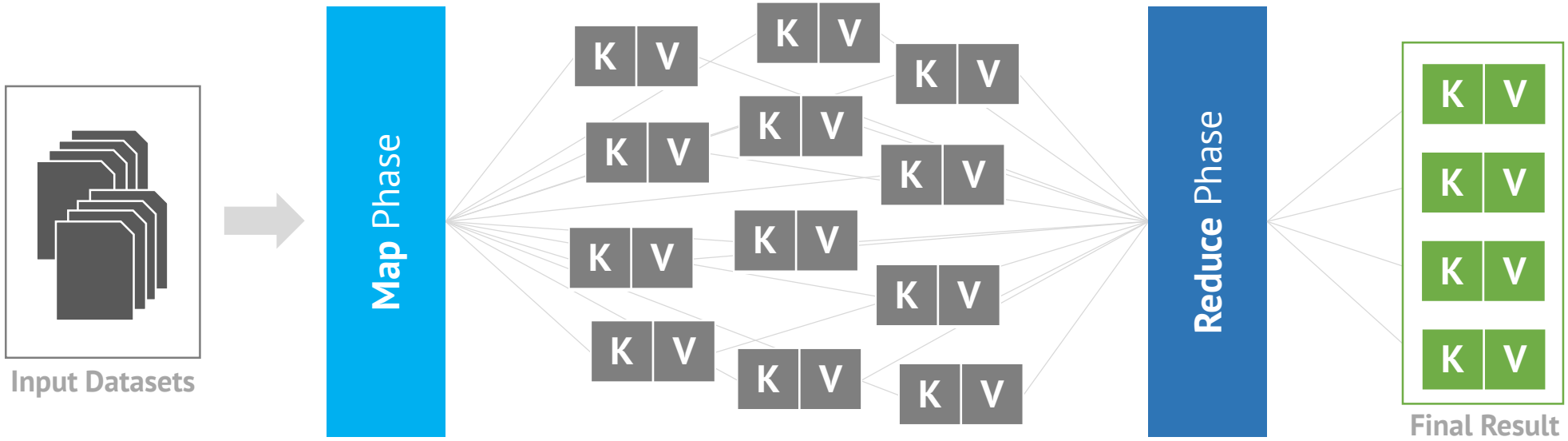
Introduction > MapReduce is trending

MapReduce has become one of the preferred programming models to hide the complexity of process and data parallelism. The power of this paradigm resides on the definition of simple `Map()` and `Reduce()` functions, that become highly-parallel operations using complex inter-processor communication:



Introduction > MapReduce is trending

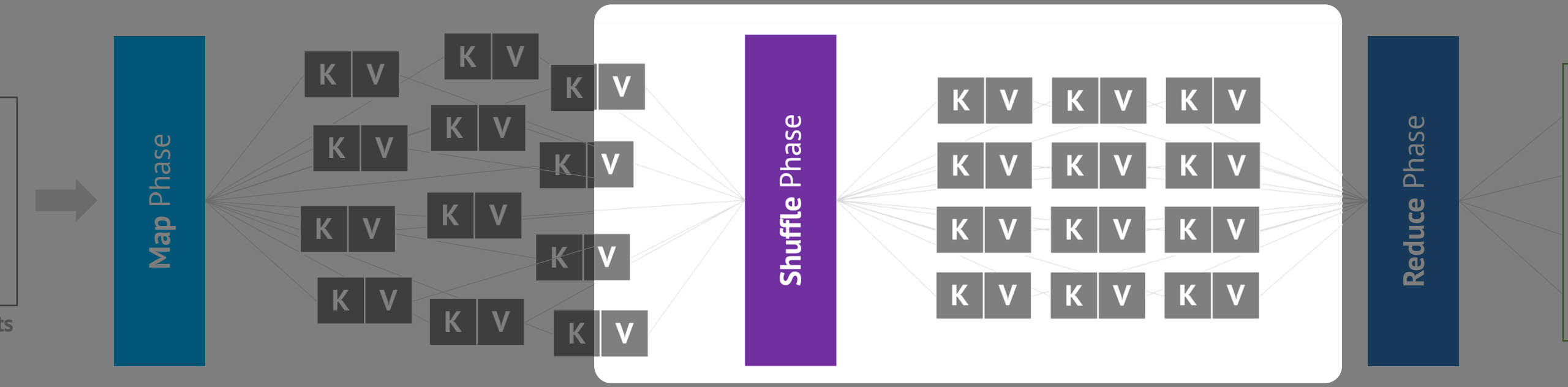
MapReduce has become one of the preferred programming models to hide the complexity of process and data parallelism. The power of this paradigm resides on the definition of simple `Map()` and `Reduce()` functions, that become highly-parallel operations using complex inter-processor communication:



Introduction > MapReduce is trending

Each **<key,value>** tuple is ordered within an intermediate Shuffle phase

Ma **of process and data parallelism**. The power of this paradigm resides on the definition of simple `Map()` and `Reduce()` functions, that become highly-parallel operations using complex inter processor communication:

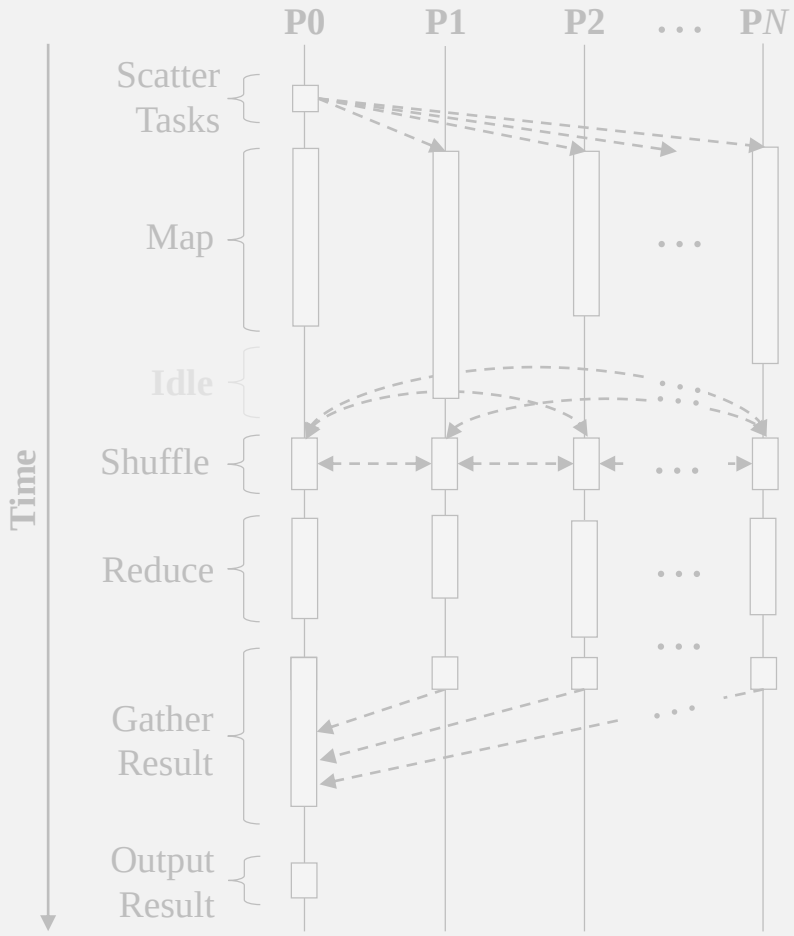


Introduction > MapReduce on HPC

Traditional MapReduce frameworks (e.g., Hadoop MapReduce) **pose numerous constraints on current supercomputers** due to the memory / storage requirements and job scheduling.

As a consequence, **MapReduce implementations on HPC are integrated using state-of-the-art MPI functionality**:

- Tasks are distributed employing a master-slave approach with scatter operations (e.g., `MPI_Scatter`).
- Fixed-length, associative key-values can be used to take advantage of reduce operations (e.g., `MPI_Reduce`).
- Intermediate data-shuffling can be mapped to collective all-to-all operations (e.g., `MPI_Alltoall`).
- When reading the input, MPI collective I/O can be used to decrease the overhead of accessing parallel file systems.

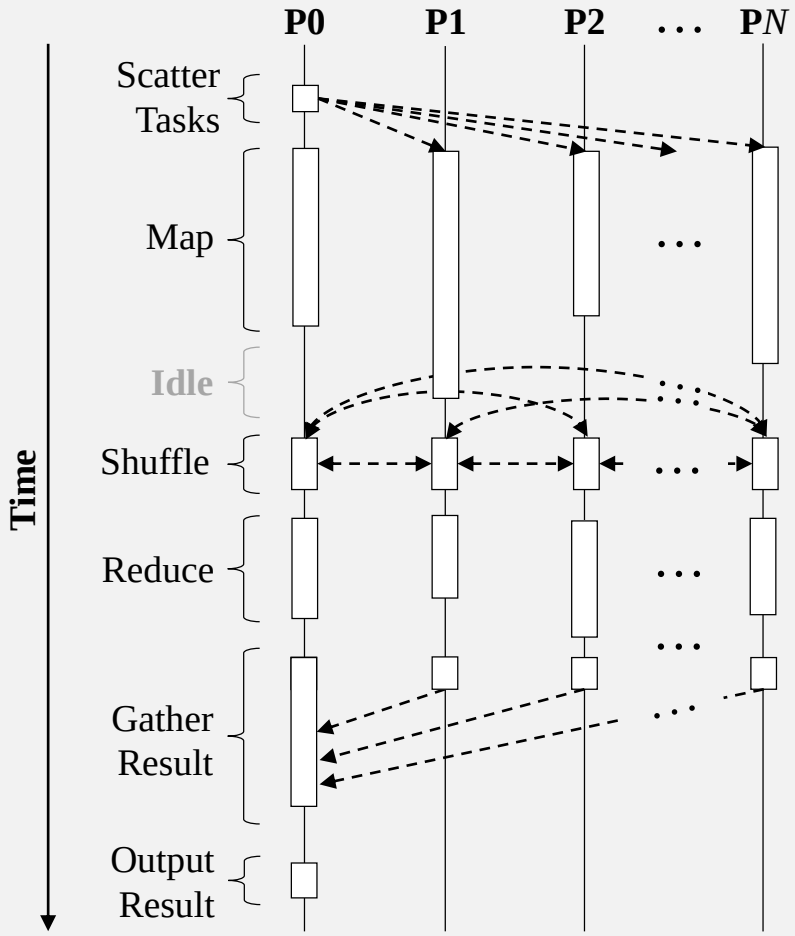


Introduction > MapReduce on HPC

Traditional MapReduce frameworks (e.g., Hadoop MapReduce) **pose numerous constraints on current supercomputers** due to the memory / storage requirements and job scheduling.

As a consequence, **MapReduce implementations on HPC are integrated using state-of-the-art MPI functionality**:

- Tasks are distributed employing a master-slave approach with scatter operations (e.g., `MPI_Scatter`).
- Fixed-length, associative key-values can be used to take advantage of reduce operations (e.g., `MPI_Reduce`).
- Intermediate data-shuffling can be mapped to collective all-to-all operations (e.g., `MPI_Alltoall`).
- When reading the input, MPI collective I/O can be used to decrease the overhead of accessing parallel file systems.

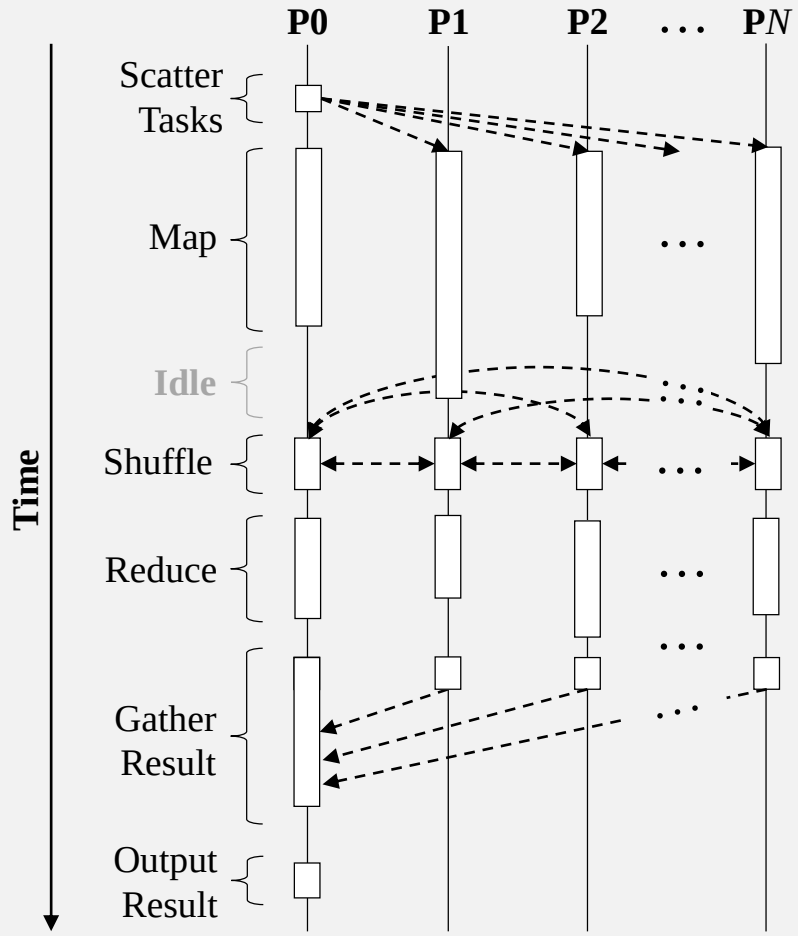


Introduction > MapReduce on HPC

Traditional MapReduce frameworks (e.g., Hadoop MapReduce) **pose numerous constraints on current supercomputers** due to the memory / storage requirements and job scheduling.

As a consequence, **MapReduce implementations on HPC are integrated using state-of-the-art MPI functionality**:

- Tasks are distributed employing a master-slave approach with scatter operations (e.g., `MPI_Scatter`).
- Fixed-length, associative key-values can be used to take advantage of reduce operations (e.g., `MPI_Reduce`).
- Intermediate data-shuffling can be mapped to collective all-to-all operations (e.g., `MPI_Alltoall`).
- When reading the input, MPI collective I/O can be used to decrease the overhead of accessing parallel file systems.

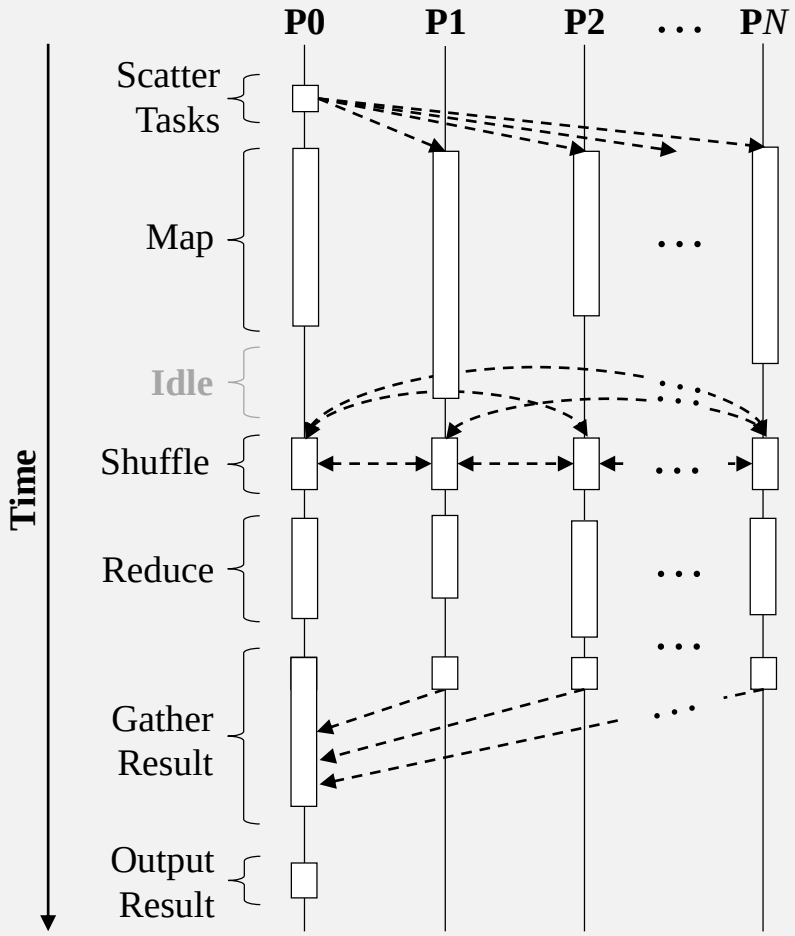


Introduction > MapReduce on HPC

Traditional MapReduce frameworks (e.g., Hadoop MapReduce) **pose numerous constraints on current supercomputers** due to the memory / storage requirements and job scheduling.

As a consequence, **MapReduce implementations on HPC are integrated using state-of-the-art MPI functionality**:

- Tasks are distributed employing a master-slave approach with scatter operations (e.g., `MPI_Scatter`).
- Fixed-length, associative key-values can be used to take advantage of reduce operations (e.g., `MPI_Reduce`).
- Intermediate data-shuffling can be mapped to collective all-to-all operations (e.g., `MPI_Alltoall`).
- When reading the input, MPI collective I/O can be used to decrease the overhead of accessing parallel file systems.

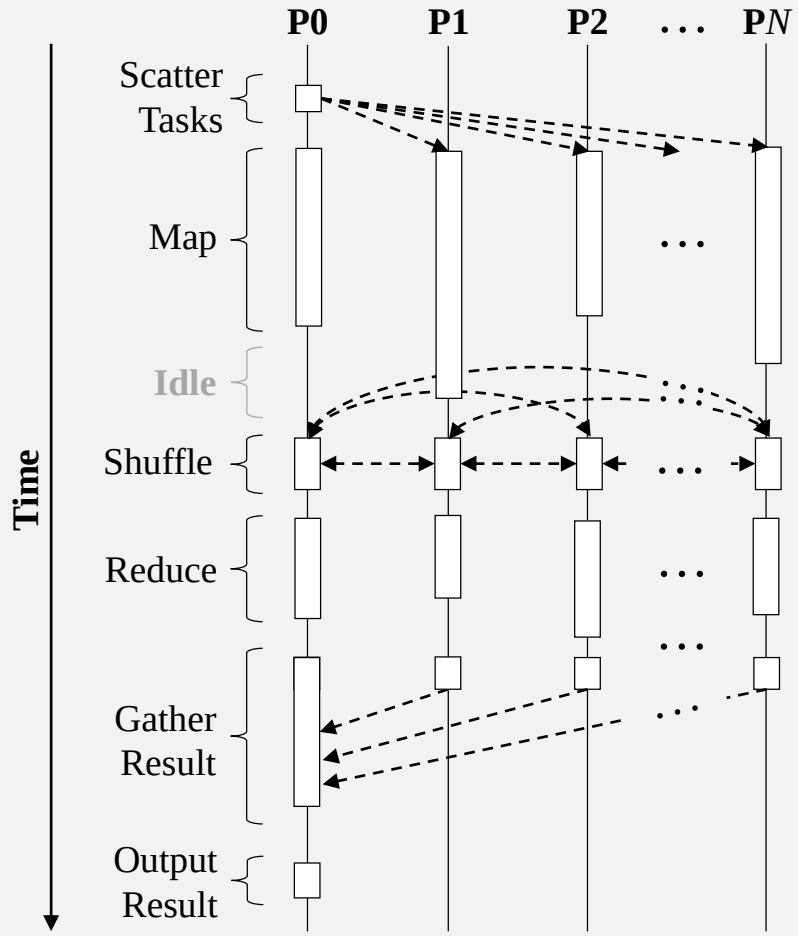


Introduction > MapReduce on HPC

Traditional MapReduce frameworks (e.g., Hadoop MapReduce) **pose numerous constraints on current supercomputers** due to the memory / storage requirements and job scheduling.

As a consequence, **MapReduce implementations on HPC are integrated using state-of-the-art MPI functionality**:

- Tasks are distributed employing a master-slave approach with scatter operations (e.g., `MPI_Scatter`).
- Fixed-length, associative key-values can be used to take advantage of reduce operations (e.g., `MPI_Reduce`).
- Intermediate data-shuffling can be mapped to collective all-to-all operations (e.g., `MPI_Alltoall`).
- When reading the input, MPI collective I/O can be used to decrease the overhead of accessing parallel file systems.



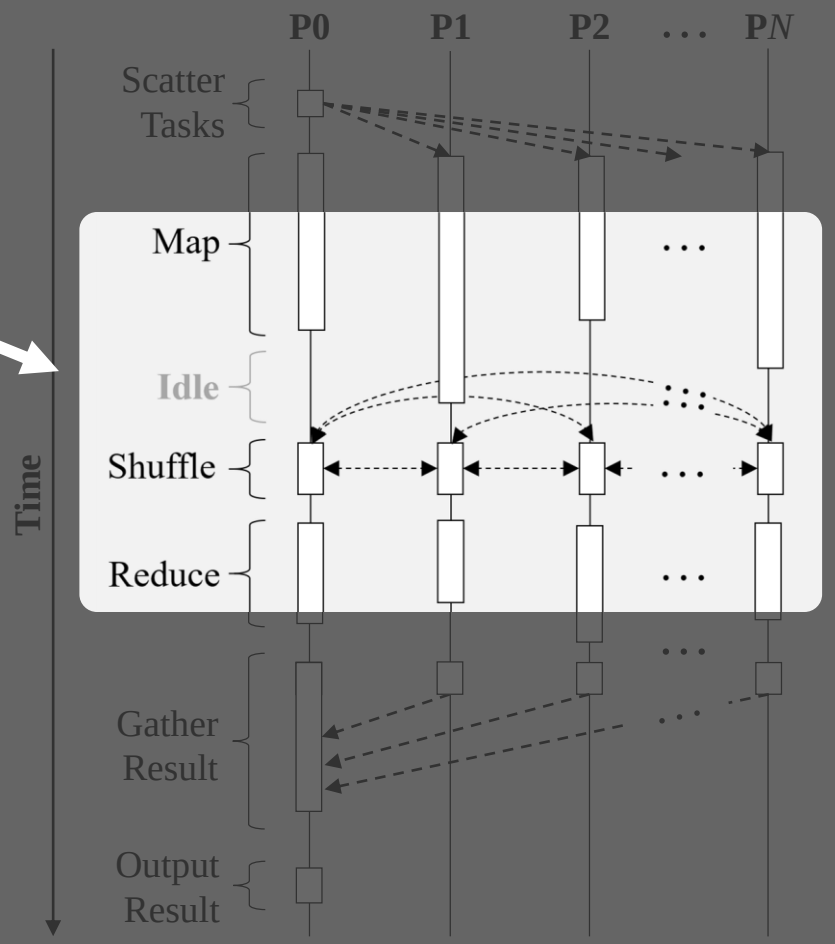
Introduction > MapReduce on HPC

Traditiona
pose num
the memory / storage requirements and job scheduling.

Idle periods can appear if the workload per process becomes unbalanced

As a consequence, **MapReduce implementations on HPC are integrated using state-of-the-art MPI functionality**:

- Tasks are distributed employing a master-slave approach with scatter operations (e.g., `MPI_Scatter`).
- Fixed-length, associative key-values can be used to take advantage of reduce operations (e.g., `MPI_Reduce`).
- Intermediate data-shuffling can be mapped to collective all-to-all operations (e.g., `MPI_Alltoall`).
- When reading the input, MPI collective I/O can be used to decrease the overhead of accessing parallel file systems.



Problem Statement

The inherent coupling between the Map and Reduce phases, can introduce constraints on highly-parallel jobs that become unexpectedly unbalanced

Introduction > Solving the challenges of MapReduce

As the concurrency of upcoming HPC clusters is expected to increase, **several limitations arise from the use of master-slave or the inherent coupling between the Map and Reduce phases of MapReduce frameworks**. These design considerations pose performance restrictions when the workload per process becomes unbalanced:

Objective #1

Decentralized Algorithm

+

Objective #2

MPI One-Sided

+

Objective #3

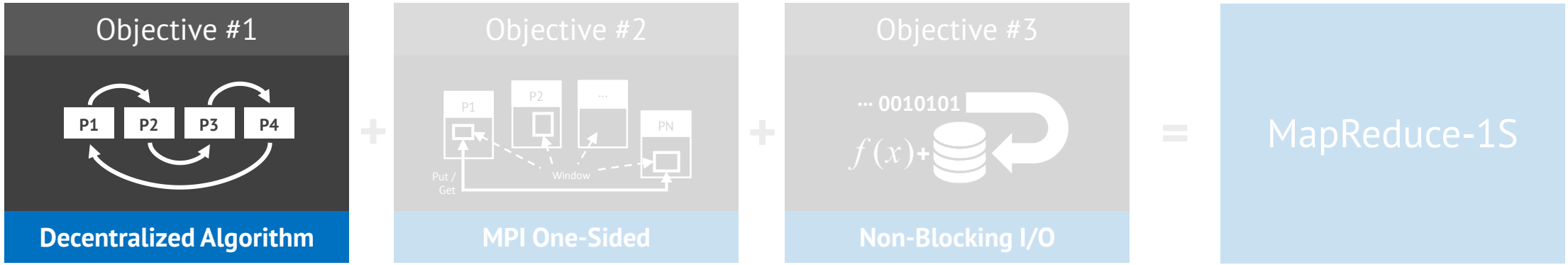
Non-Blocking I/O

=

MapReduce-1S

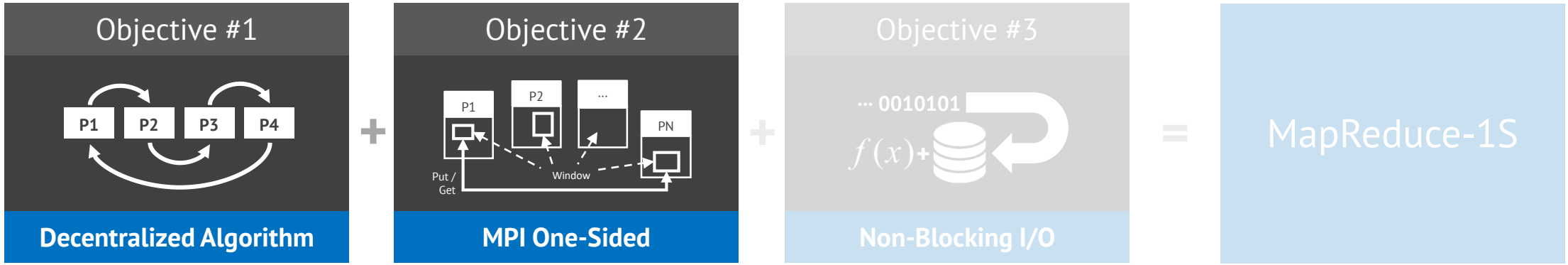
Introduction > Solving the challenges of MapReduce

As the concurrency of upcoming HPC clusters is expected to increase, **several limitations arise from the use of master-slave or the inherent coupling between the Map and Reduce phases of MapReduce frameworks**. These design considerations pose performance restrictions when the workload per process becomes unbalanced:



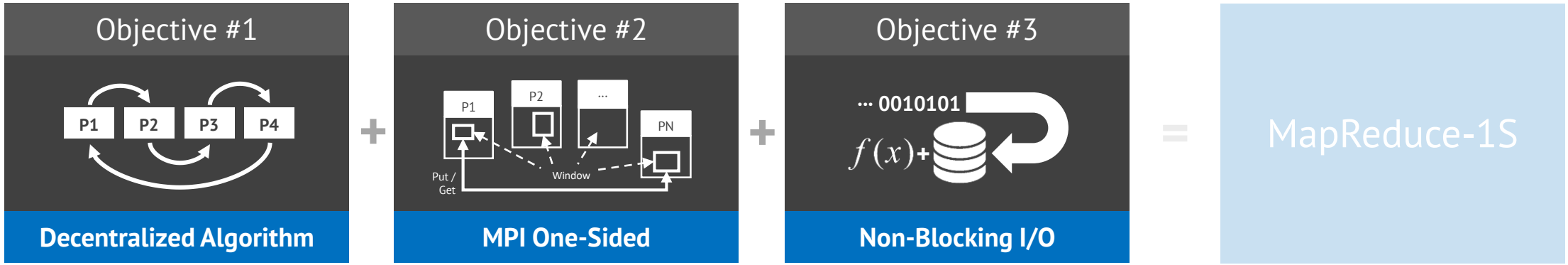
Introduction > Solving the challenges of MapReduce

As the concurrency of upcoming HPC clusters is expected to increase, **several limitations arise from the use of master-slave or the inherent coupling between the Map and Reduce phases of MapReduce frameworks**. These design considerations pose performance restrictions when the workload per process becomes unbalanced:



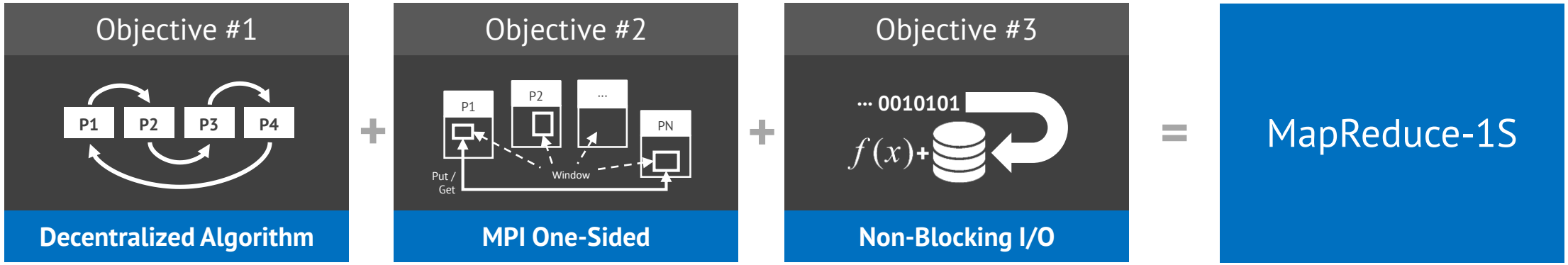
Introduction > Solving the challenges of MapReduce

As the concurrency of upcoming HPC clusters is expected to increase, **several limitations arise from the use of master-slave or the inherent coupling between the Map and Reduce phases of MapReduce frameworks**. These design considerations pose performance restrictions when the workload per process becomes unbalanced:



Introduction > Solving the challenges of MapReduce

As the concurrency of upcoming HPC clusters is expected to increase, **several limitations arise from the use of master-slave or the inherent coupling between the Map and Reduce phases of MapReduce frameworks**. These design considerations pose performance restrictions when the workload per process becomes unbalanced:



MapReduce-1S / MapReduce "One-Sided"

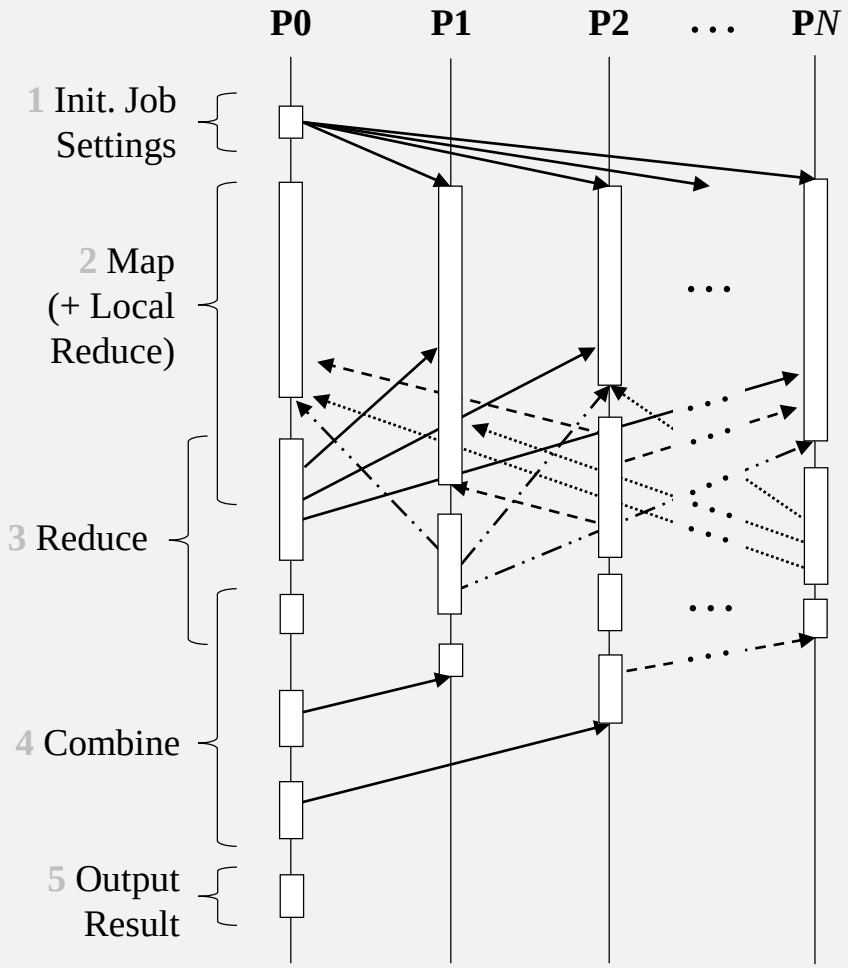
Integrating the MPI one-sided communication model and non-blocking I/O to reduce the overhead on imbalanced workloads

Methodology > Phases

MapReduce-1S inherits the core principles of traditional MapReduce frameworks, such as Hadoop MapReduce.

We divide the execution into four different isolated phases:

I. **Map**. Transforms a given input into multiple key-value pairs. Each key-value is assigned to a process (hash).
II. **Local Reduce**. Aggregates certain key-value pairs locally, whenever possible, to reduce constraints.
III. **Reduce**. Aggregates the key-value pairs found by the rest of the processes, using one-sided operations.
IV. **Combine**. Combines the aggregated key-value pairs to generate the final result (similar to Shuffle).

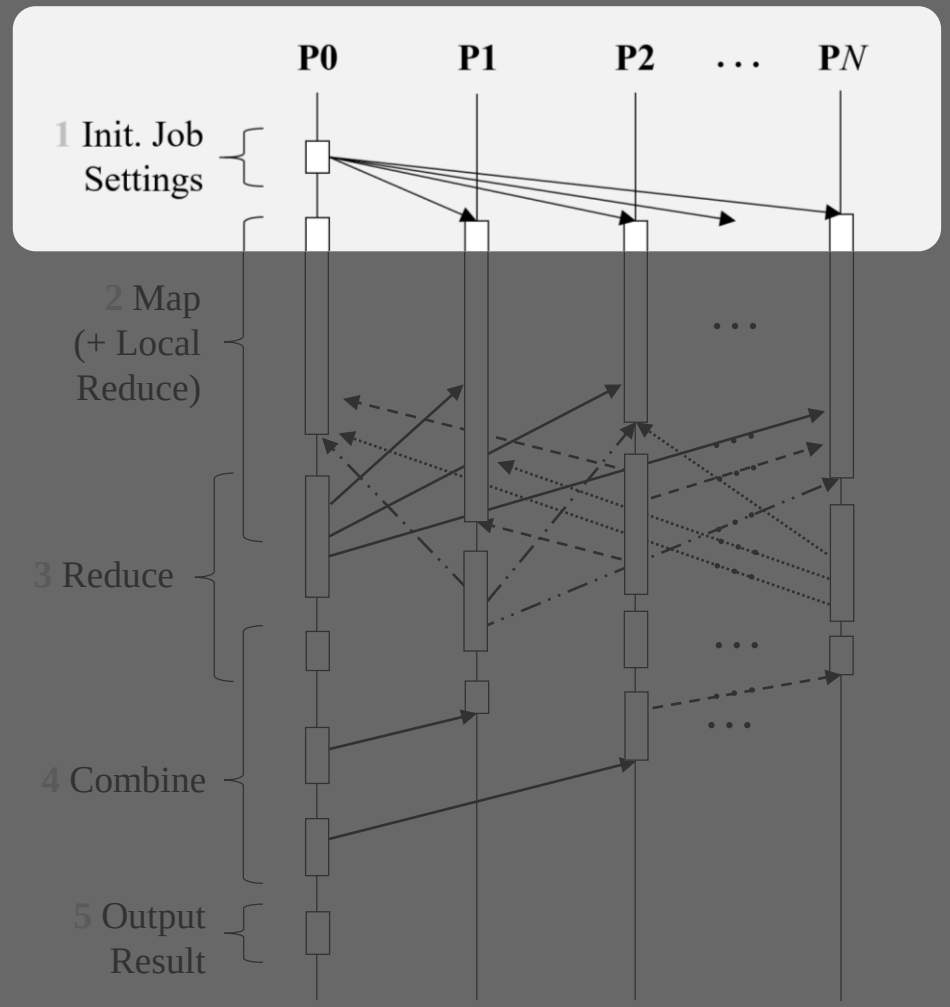


Methodology > Phases

MapReduce-1S inherits the core principles of traditional MapReduce frameworks, such as Hadoop MapReduce.

We divide the execution into four different isolated phases:

I. **Map**. Transforms a given input into multiple key-value pairs. Each key-value is assigned to a process (hash).
II. **Local Reduce**. Aggregates certain key-value pairs locally, whenever possible, to reduce constraints.
III. **Reduce**. Aggregates the key-value pairs found by the rest of the processes, using one-sided operations.
IV. **Combine**. Combines the aggregated key-value pairs to generate the final result (similar to Shuffle).



Methodology > Phases

MapReduce-1S inherits the core principles of traditional MapReduce frameworks, such as Hadoop MapReduce.

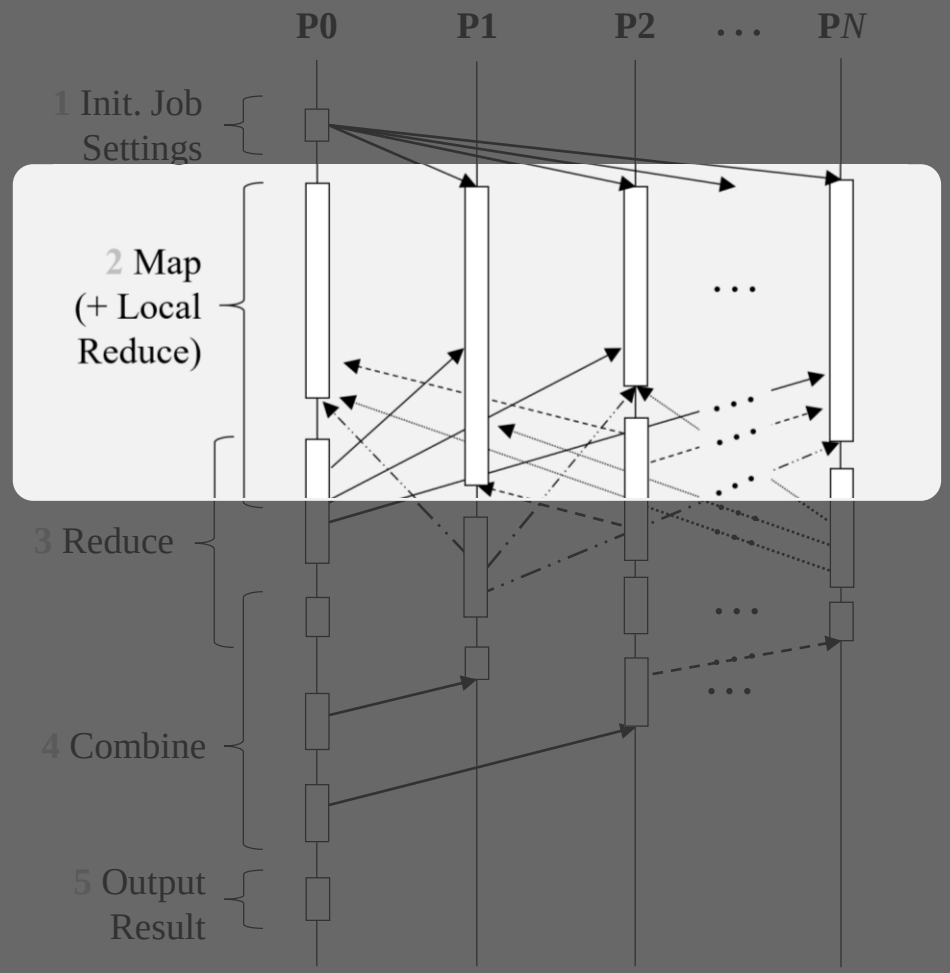
We divide the execution into four different isolated phases:

I. **Map**. Transforms a given input into multiple key-value pairs. Each key-value is assigned to a process (hash).

II. **Local Reduce**. Aggregates certain key-value pairs locally, whenever possible, to reduce constraints.

III. **Reduce**. Aggregates the key-value pairs found by the rest of the processes, using one-sided operations.

IV. **Combine**. Combines the aggregated key-value pairs to generate the final result (similar to Shuffle).



Methodology > Phases

MapReduce-1S inherits the core principles of traditional MapReduce frameworks, such as Hadoop MapReduce.

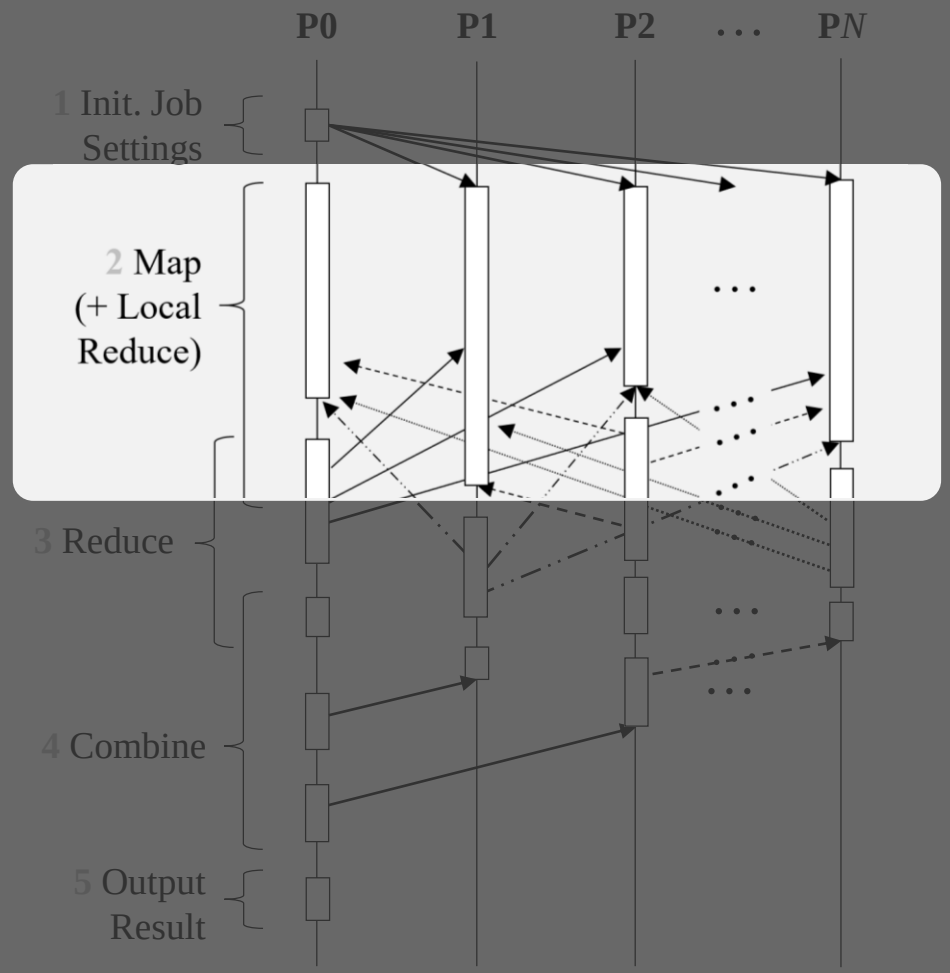
We divide the execution into four different isolated phases:

I. **Map**. Transforms a given input into multiple key-value pairs. Each key-value is assigned to a process (hash).

II. **Local Reduce**. Aggregates certain key-value pairs locally, whenever possible, to reduce constraints.

III. **Reduce**. Aggregates the key-value pairs found by the rest of the processes, using one-sided operations.

IV. **Combine**. Combines the aggregated key-value pairs to generate the final result (similar to Shuffle).



Methodology > Phases

MapReduce-1S inherits the core principles of traditional MapReduce frameworks, such as Hadoop MapReduce.

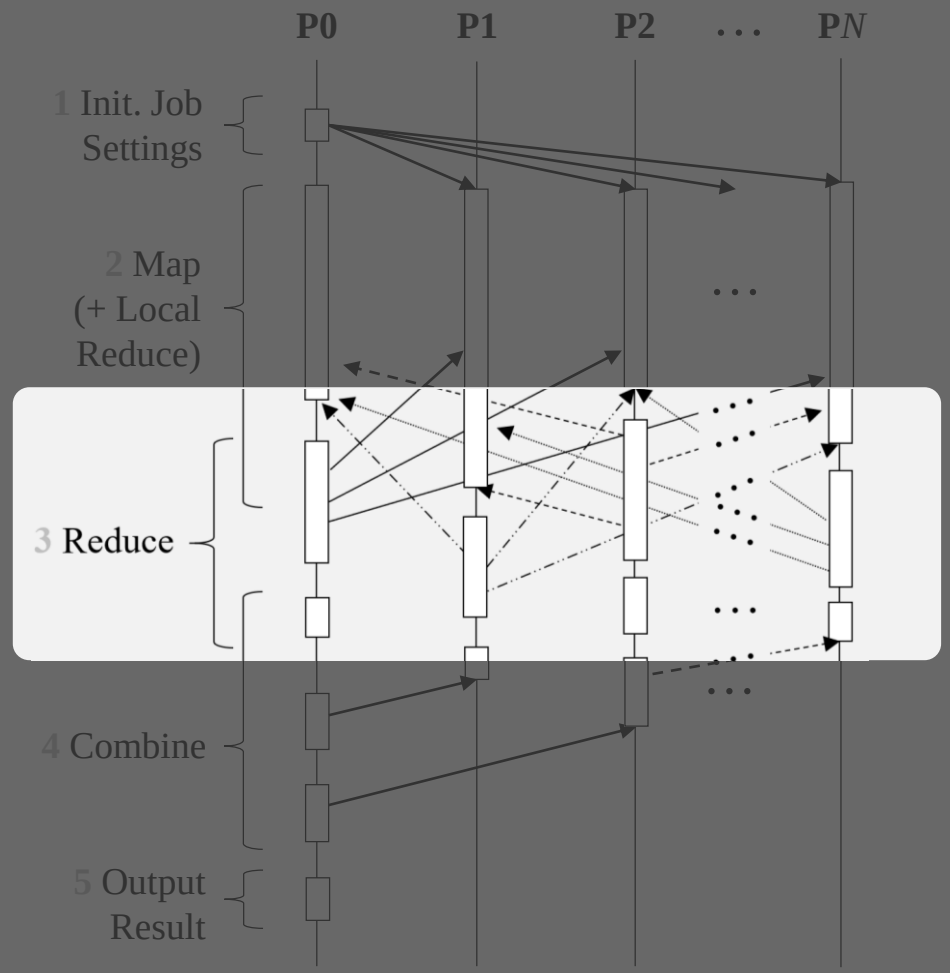
We divide the execution into four different isolated phases:

I. **Map**. Transforms a given input into multiple key-value pairs. Each key-value is assigned to a process (hash).

II. **Local Reduce**. Aggregates certain key-value pairs locally, whenever possible, to reduce constraints.

III. **Reduce**. Aggregates the key-value pairs found by the rest of the processes, using one-sided operations.

IV. **Combine**. Combines the aggregated key-value pairs to generate the final result (similar to Shuffle).

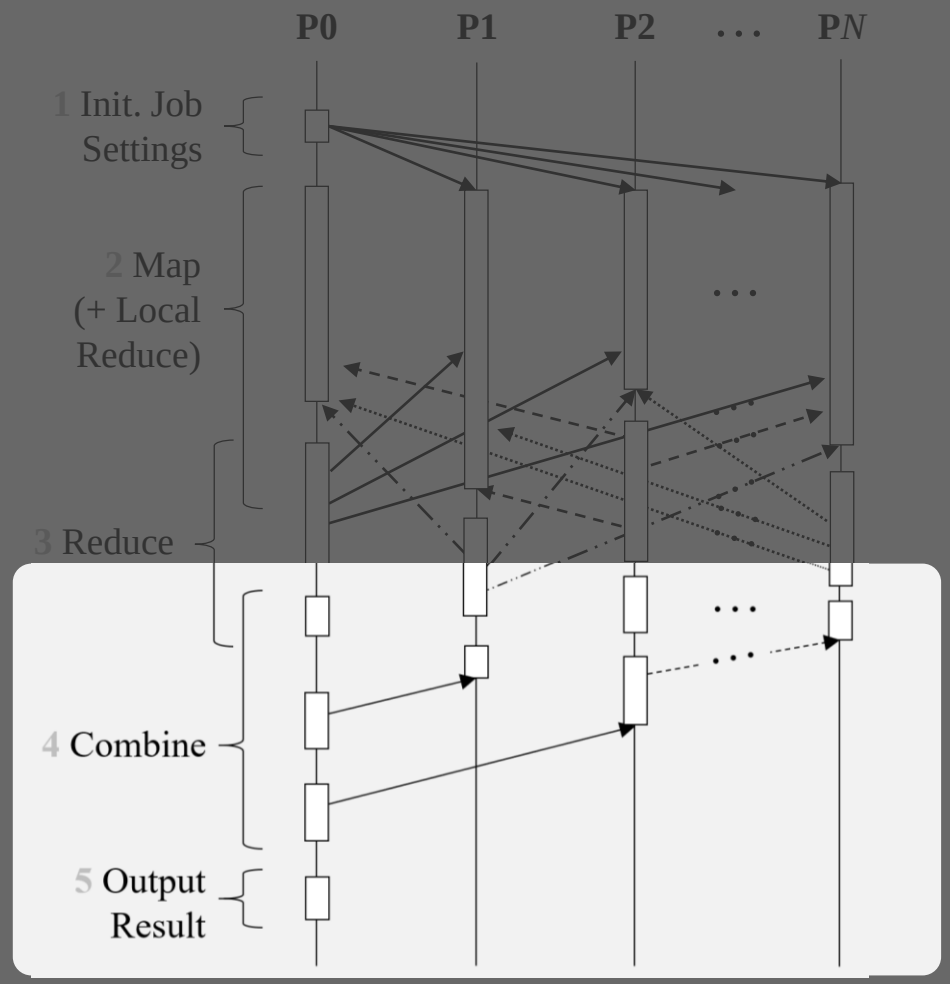


Methodology > Phases

MapReduce-1S inherits the core principles of traditional MapReduce frameworks, such as Hadoop MapReduce.

We divide the execution into four different isolated phases:

I. **Map**. Transforms a given input into multiple key-value pairs. Each key-value is assigned to a process (hash).
II. **Local Reduce**. Aggregates certain key-value pairs locally, whenever possible, to reduce constraints.
III. **Reduce**. Aggregates the key-value pairs found by the rest of the processes, using one-sided operations.
IV. **Combine**. Combines the aggregated key-value pairs to generate the final result (similar to Shuffle).

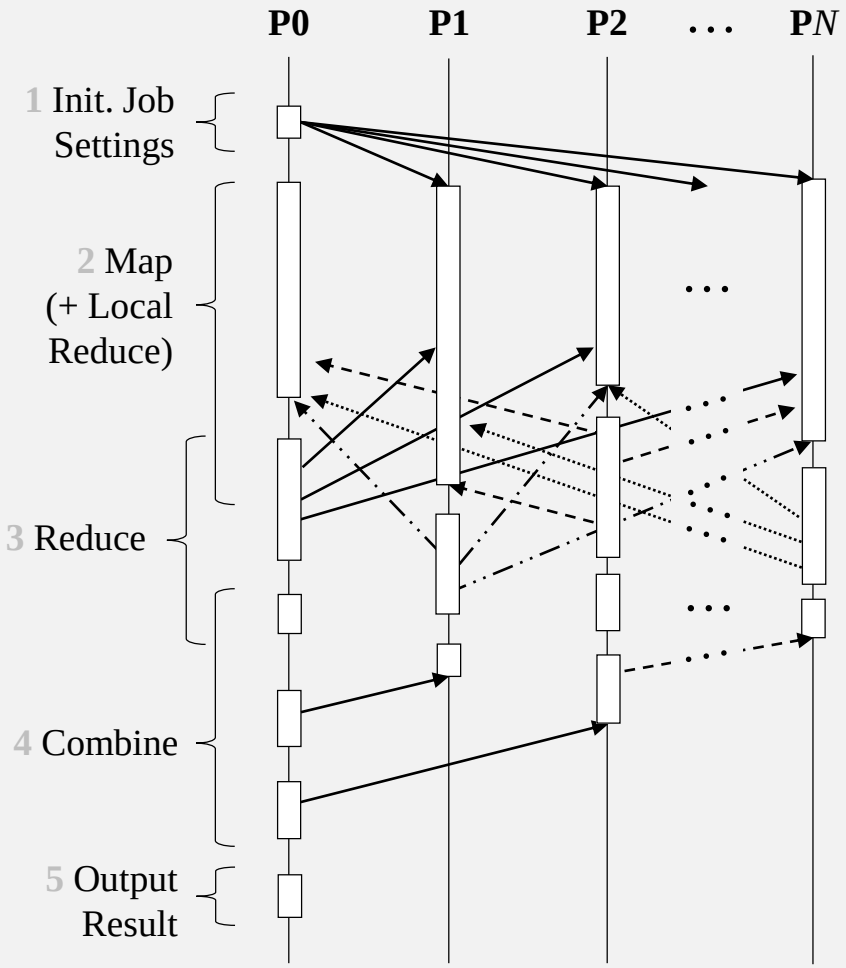


Methodology > Phases

MapReduce-1S inherits the core principles of traditional MapReduce frameworks, such as Hadoop MapReduce.

We divide the execution into four different isolated phases:

I. **Map**. Transforms a given input into multiple key-value pairs. Each key-value is assigned to a process (hash).
II. **Local Reduce**. Aggregates certain key-value pairs locally, whenever possible, to reduce constraints.
III. **Reduce**. Aggregates the key-value pairs found by the rest of the processes, using one-sided operations.
IV. **Combine**. Combines the aggregated key-value pairs to generate the final result (similar to Shuffle).

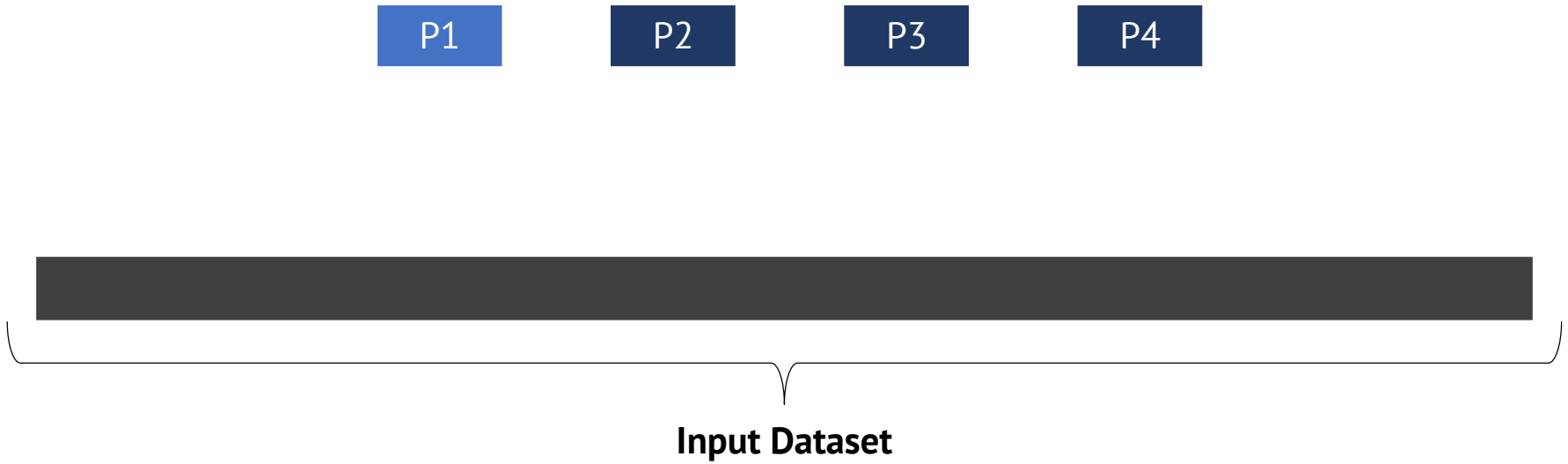


Methodology > Reading Inputs with Non-Blocking I/O

The input datasets are split into equally-sized tasks, that are later handled in parallel by each process. Instead of following a master-slave approach, **each process decide the next task to perform based on the rank, task size, and file offset between tasks**. Thus, we decentralize the MapReduce algorithm:

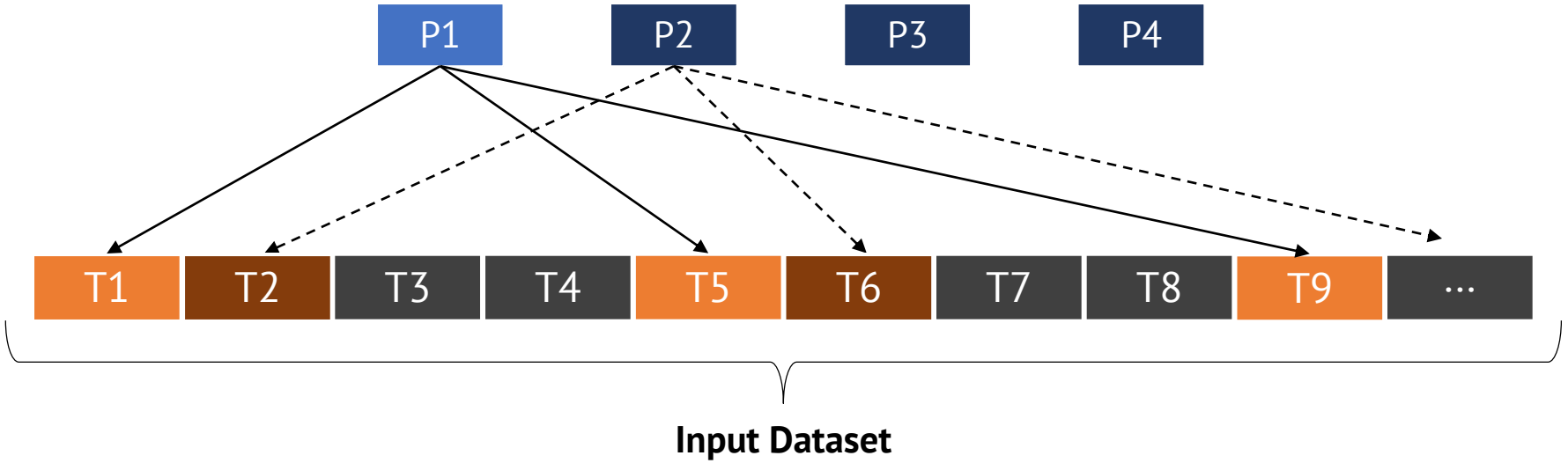
Methodology > Reading Inputs with Non-Blocking I/O

The input datasets are split into equally-sized tasks, that are later handled in parallel by each process. Instead of following a master-slave approach, **each process decide the next task to perform based on the rank, task size, and file offset between tasks**. Thus, we decentralize the MapReduce algorithm:



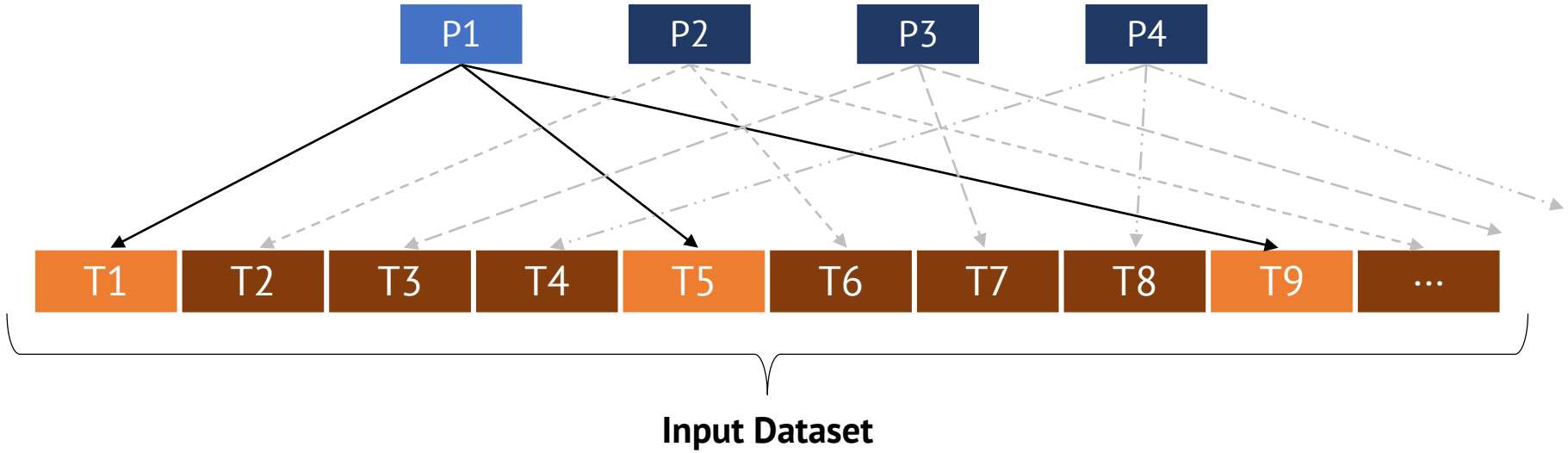
Methodology > Reading Inputs with Non-Blocking I/O

The input datasets are split into equally-sized tasks, that are later handled in parallel by each process. Instead of following a master-slave approach, **each process decide the next task to perform based on the rank, task size, and file offset between tasks**. Thus, we decentralize the MapReduce algorithm:



Methodology > Reading Inputs with Non-Blocking I/O

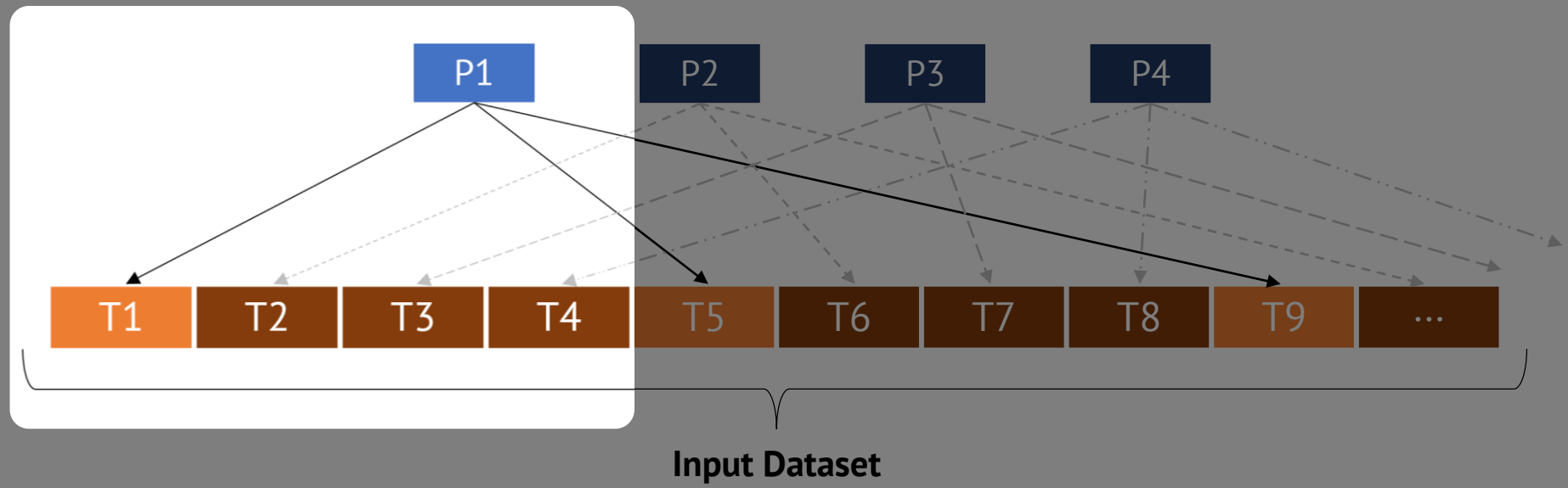
The input datasets are split into equally-sized tasks, that are later handled in parallel by each process. Instead of following a master-slave approach, **each process decide the next task to perform based on the rank, task size, and file offset between tasks**. Thus, we decentralize the MapReduce algorithm:



Methodology > Reading Inputs with Non-Blocking I/O

The input datasets are split into equally-sized tasks, that are later handled in parallel by each process. Instead of following a m **d file offset between tasks**. Thus, we decentralize the MapReduce algorithm:

The input portion for the task is retrieved individually using non-blocking MPI I/O

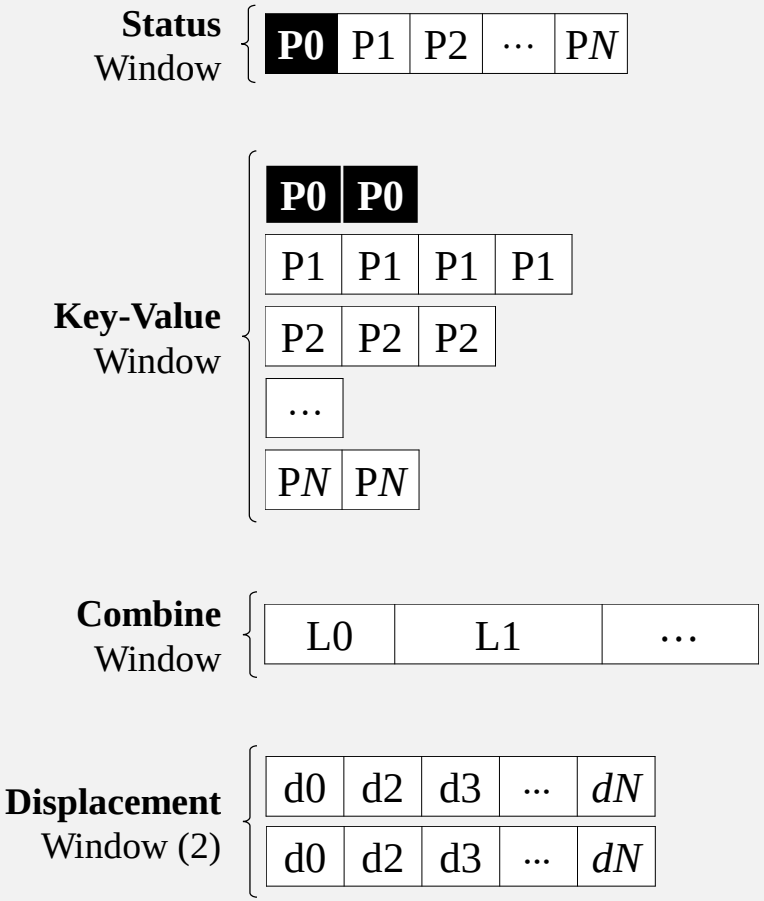


Methodology > MPI Windows

In order to enable remote memory communications during the execution phases, we define a multi-window configuration.

In particular, **four different windows per process are required**:

- **"Status" Window**. Defines the current status for each individual process. The status is updated remotely.
- **"Key-Value" Window.** This multi-dimensional, dynamic window contains buckets to store the key-value pairs, indexed by the target rank.
- **"Combine" Window.** Designed for the Combine phase, it contains an ordered single-dimension, dynamic window.
- **"Displacement" Window.** Two additional displacement windows are defined to support the others.

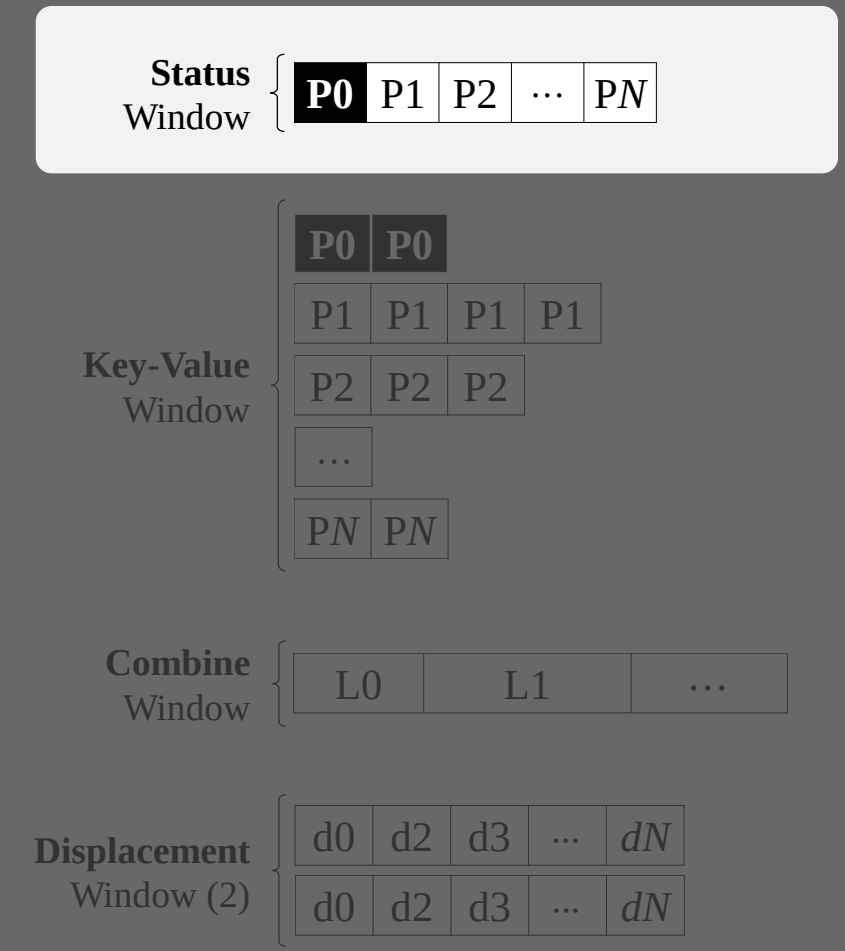


Methodology > MPI Windows

In order to enable remote memory communications during the execution phases, we define a multi-window configuration.

In particular, **four different windows per process are required:**

- **"Status" Window**. Defines the current status for each individual process. The status is updated remotely.
- **"Key-Value" Window.** This multi-dimensional, dynamic window contains buckets to store the key-value pairs, indexed by the target rank.
- **"Combine" Window.** Designed for the Combine phase, it contains an ordered single-dimension, dynamic window.
- **"Displacement" Window.** Two additional displacement windows are defined to support the others.

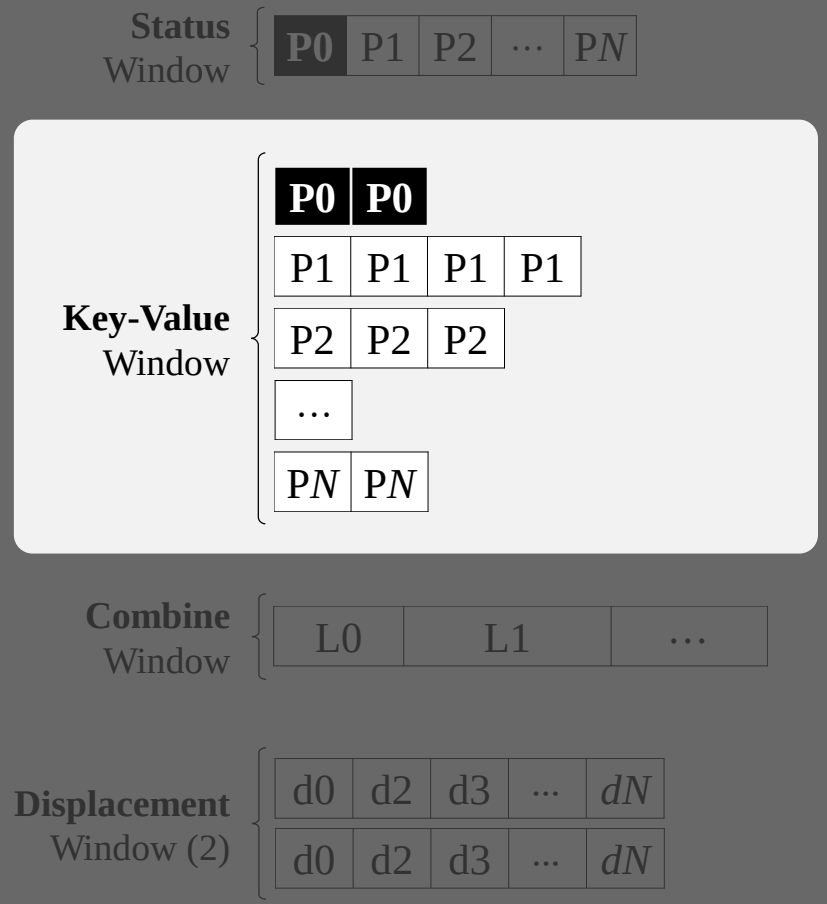


Methodology > MPI Windows

In order to enable remote memory communications during the execution phases, we define a multi-window configuration.

In particular, **four different windows per process are required:**

- **"Status" Window**. Defines the current status for each individual process. The status is updated remotely.
- **"Key-Value" Window.** This multi-dimensional, dynamic window contains buckets to store the key-value pairs, indexed by the target rank.
- **"Combine" Window.** Designed for the Combine phase, it contains an ordered single-dimension, dynamic window.
- **"Displacement" Window.** Two additional displacement windows are defined to support the others.

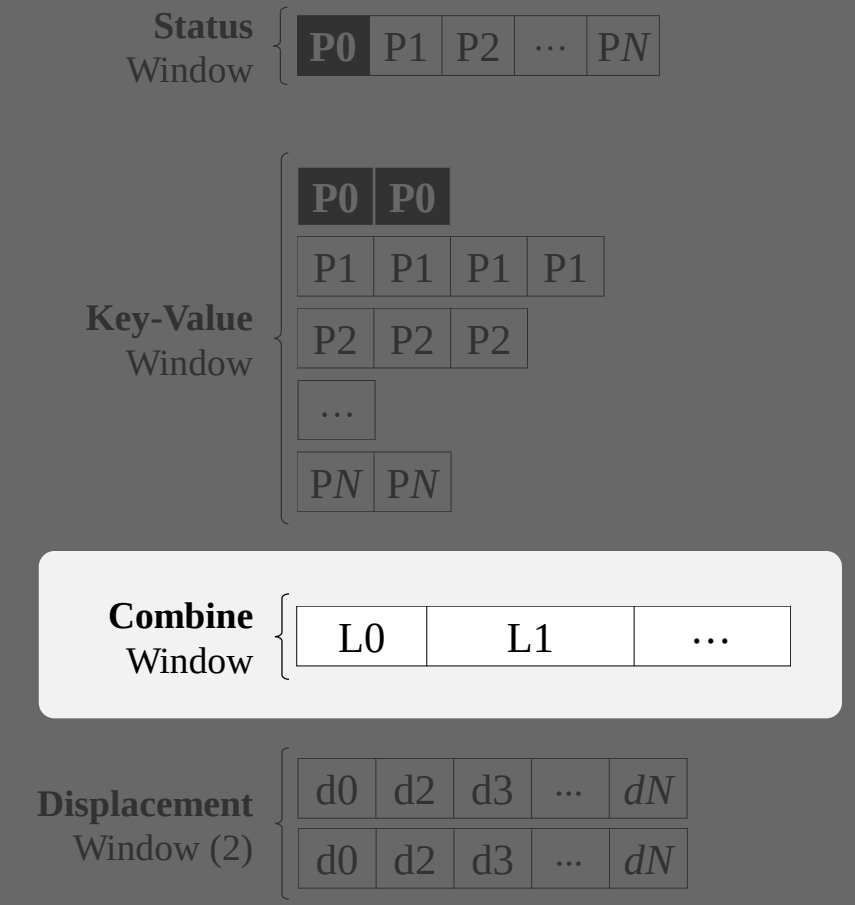


Methodology > MPI Windows

In order to enable remote memory communications during the execution phases, we define a multi-window configuration.

In particular, **four different windows per process are required:**

- **"Status" Window**. Defines the current status for each individual process. The status is updated remotely.
- **"Key-Value" Window.** This multi-dimensional, dynamic window contains buckets to store the key-value pairs, indexed by the target rank.
- **"Combine" Window.** Designed for the Combine phase, it contains an ordered single-dimension, dynamic window.
- **"Displacement" Window.** Two additional displacement windows are defined to support the others.

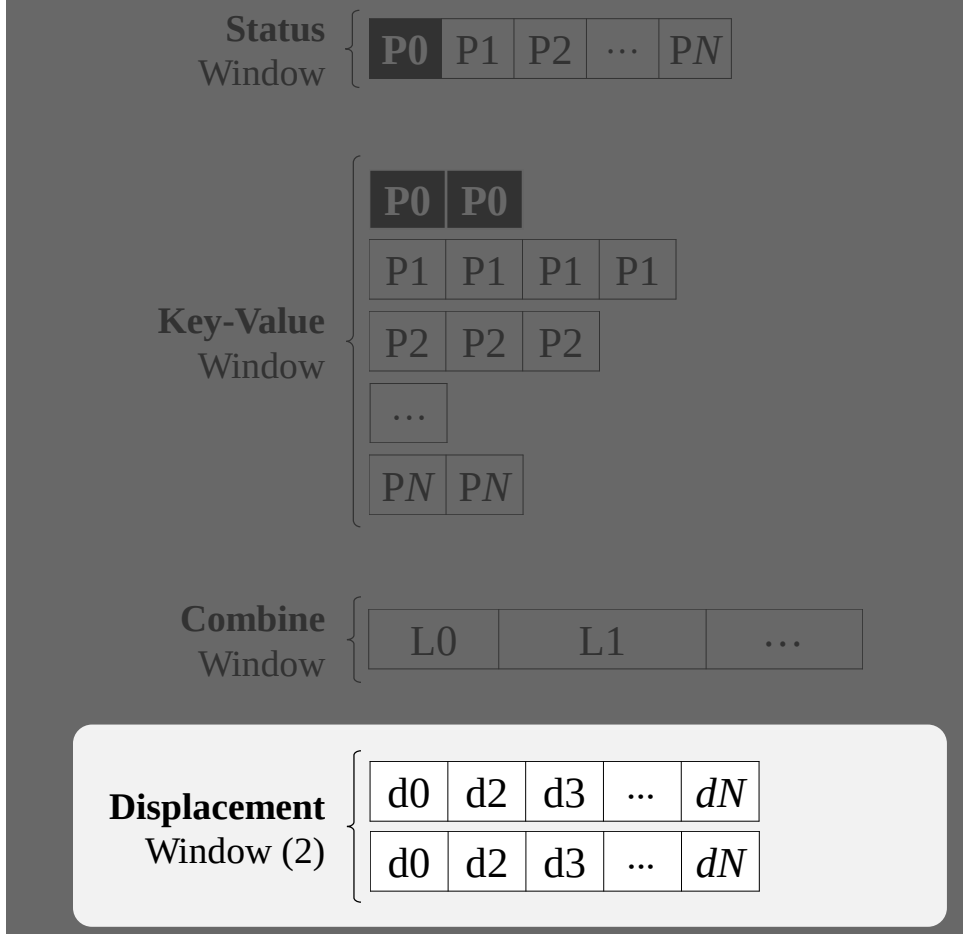


Methodology > MPI Windows

In order to enable remote memory communications during the execution phases, we define a multi-window configuration.

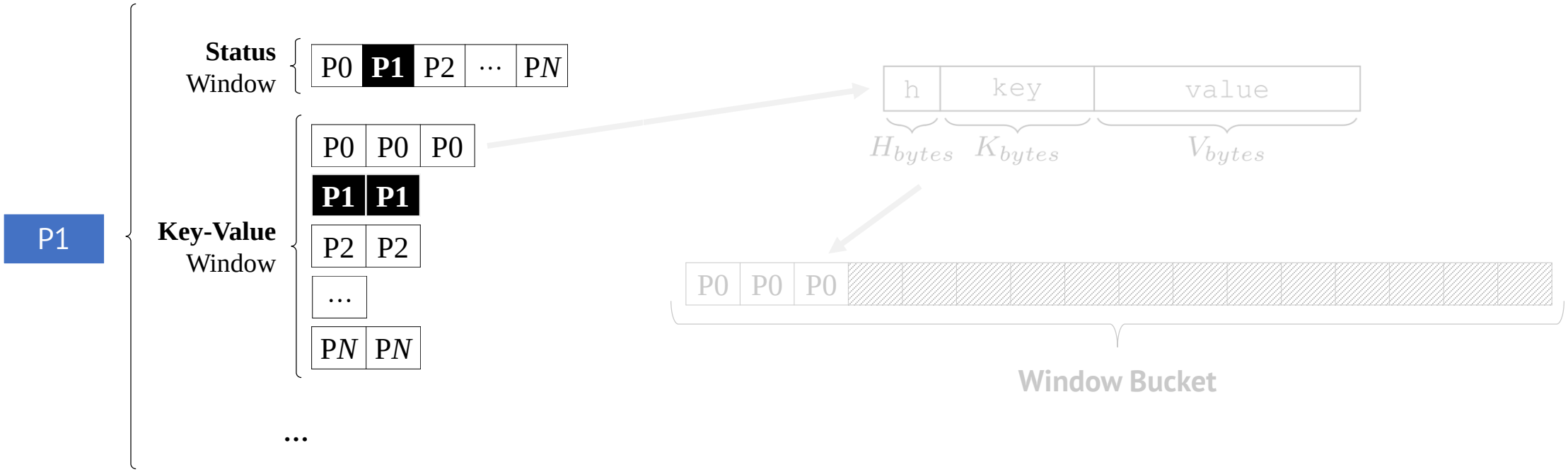
In particular, **four different windows per process are required:**

- **"Status" Window**. Defines the current status for each individual process. The status is updated remotely.
- **"Key-Value" Window.** This multi-dimensional, dynamic window contains buckets to store the key-value pairs, indexed by the target rank.
- **"Combine" Window.** Designed for the Combine phase, it contains an ordered single-dimension, dynamic window.
- **"Displacement" Window.** Two additional displacement windows are defined to support the others.



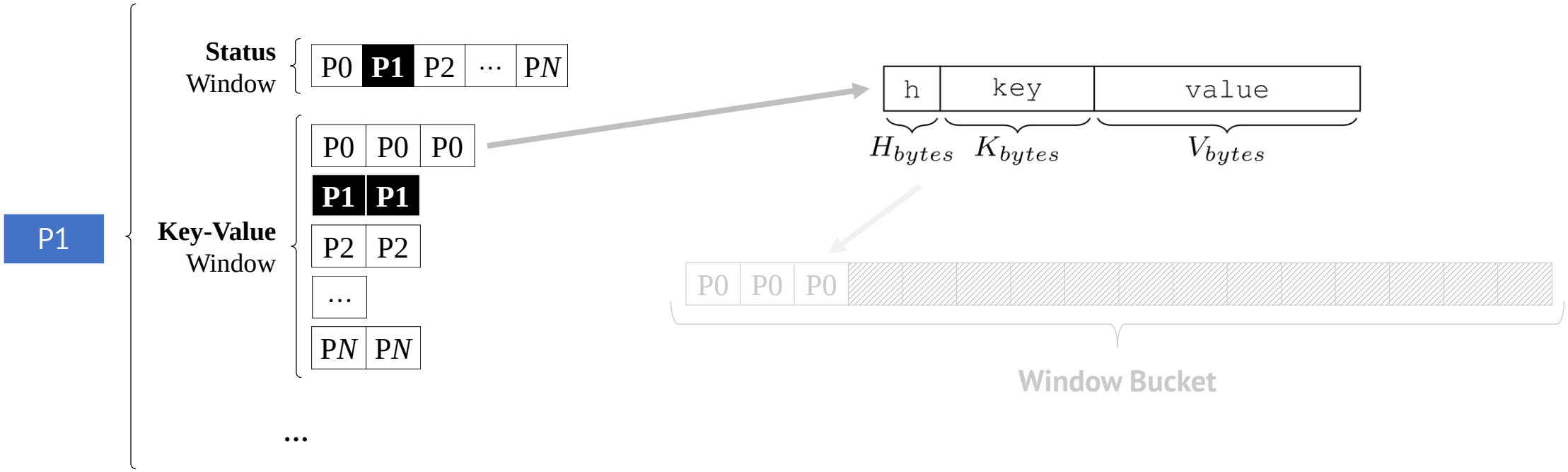
Methodology > Custom Memory Management

When a new key-value pair is found, **we use a custom memory management to store the correspondent `<key,value>` tuple**. Each key-value pair is mapped inside the current bucket assigned to the target process:



Methodology > Custom Memory Management

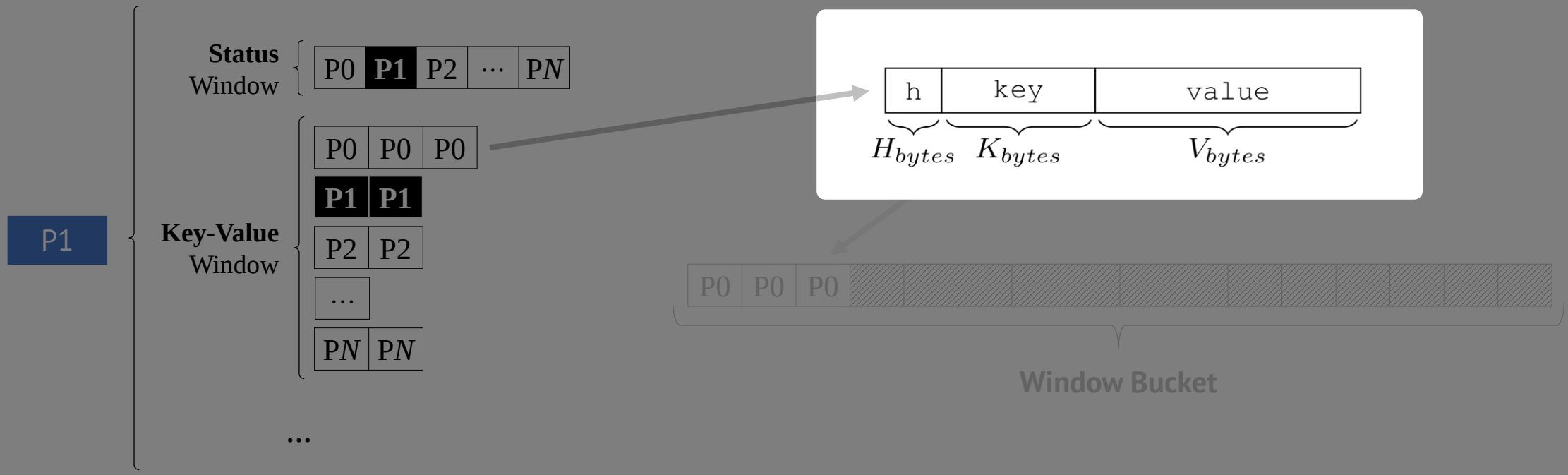
When a new key-value pair is found, **we use a custom memory management to store the correspondent `<key,value>` tuple**. Each key-value pair is mapped inside the current bucket assigned to the target process:



Methodology > Custom Memory Management

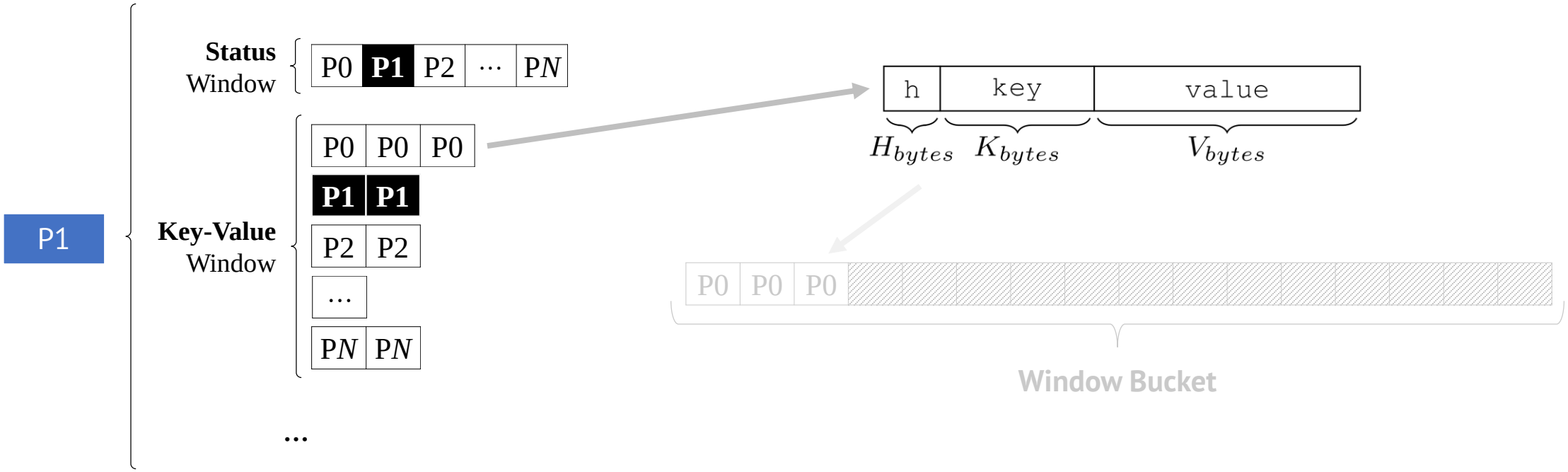
Support for variable-length **<key,value>** tuples, of arbitrary K and V bytes

When **respondent <key,value> tuple**. Each key-value pair is mapped inside the current bucket assigned to the target process:



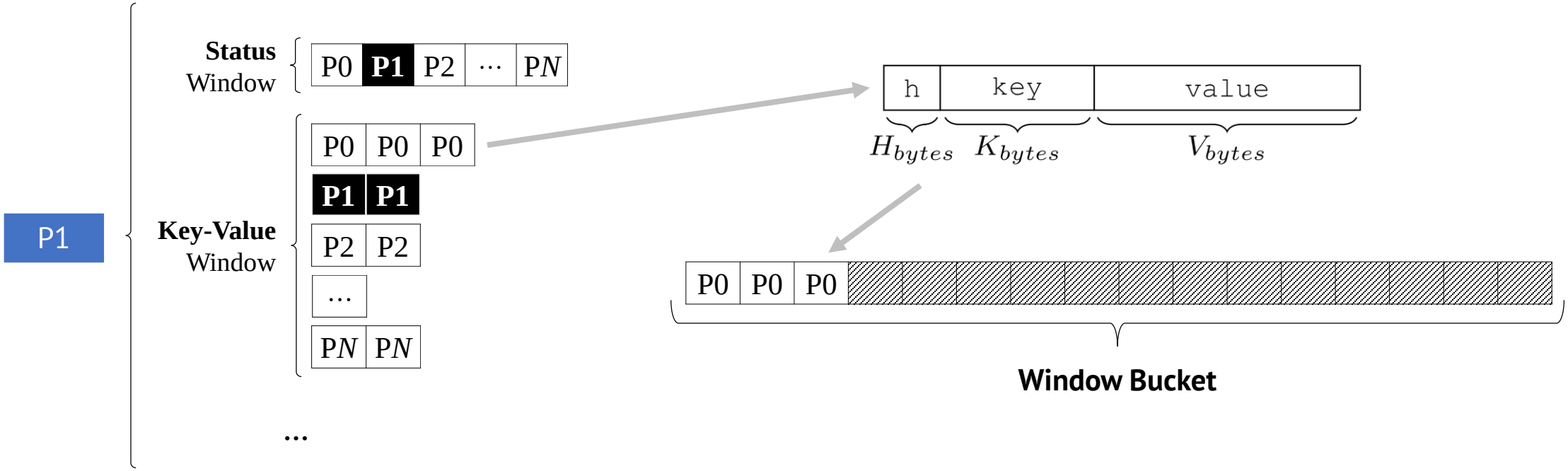
Methodology > Custom Memory Management

When a new key-value pair is found, **we use a custom memory management to store the correspondent `<key,value>` tuple**. Each key-value pair is mapped inside the current bucket assigned to the target process:



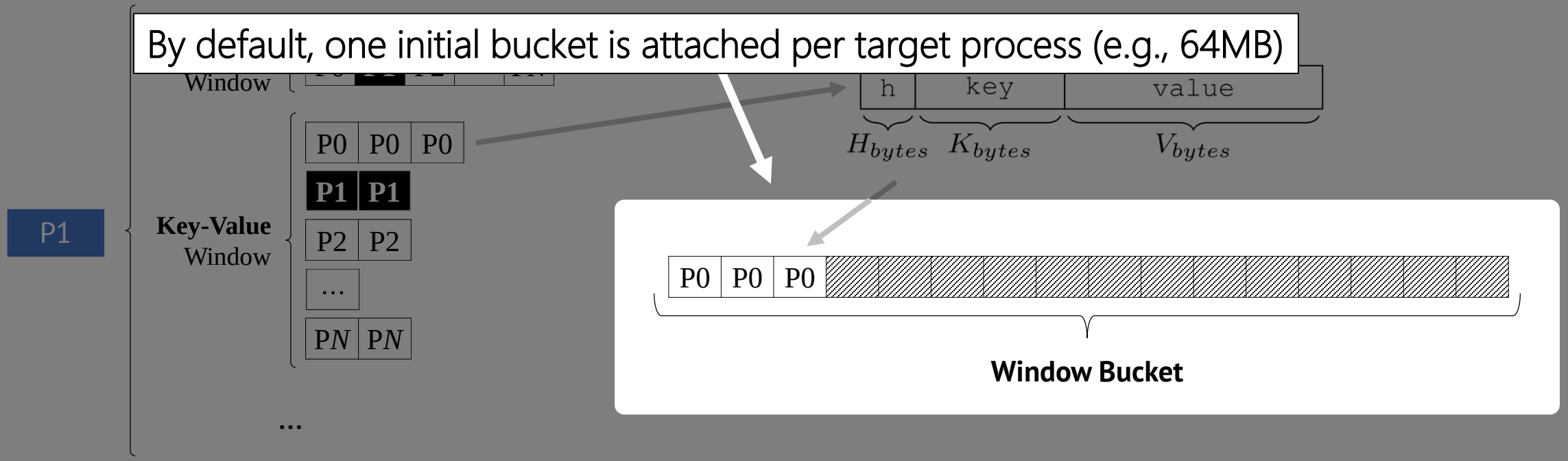
Methodology > Custom Memory Management

When a new key-value pair is found, **we use a custom memory management to store the correspondent `<key,value>` tuple**. Each key-value pair is mapped inside the current bucket assigned to the target process:



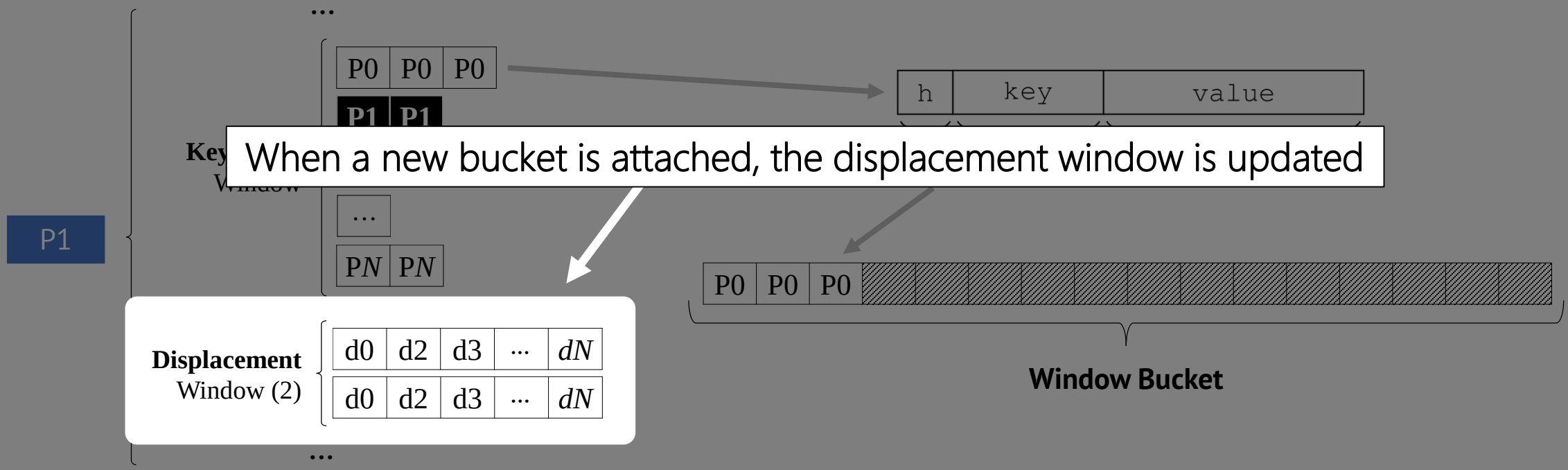
Methodology > Custom Memory Management

When a new key-value pair is found, **we use a custom memory management to store the correspondent `<key,value>` tuple**. Each key-value pair is mapped inside the current bucket assigned to the target process:



Methodology > Custom Memory Management

When a new key-value pair is found, **we use a custom memory management to store the correspondent `<key,value>` tuple**. Each key-value pair is mapped inside the current bucket assigned to the target process:



Methodology > Defining the Target Process

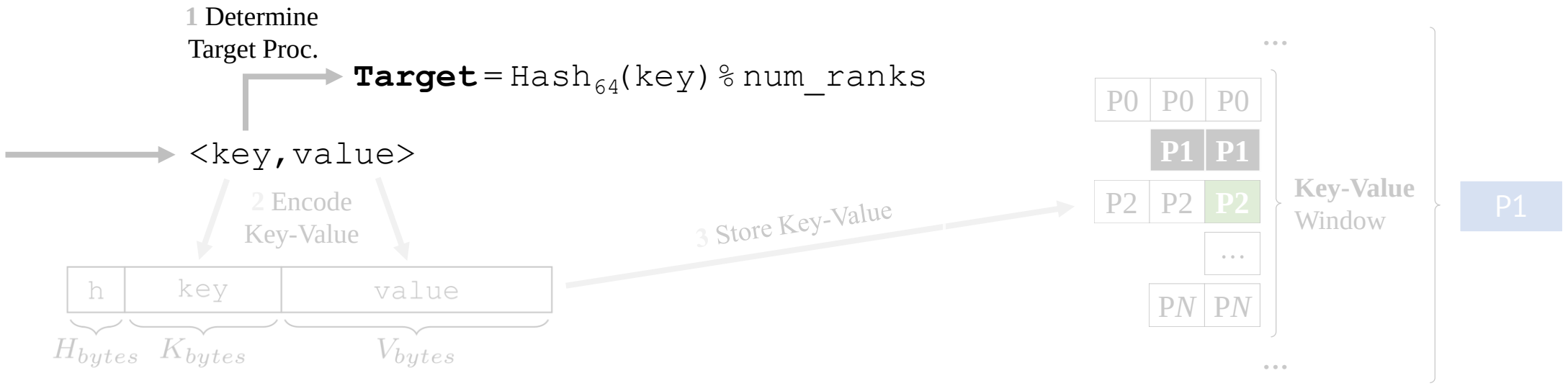
Inside the Map phase, **the target process is determined by first generating a 64-bit hash of the key.** Thereafter, a mapping to the associated chunk inside the Key-Value window is established:

Methodology > Defining the Target Process

Inside the Map phase, **the target process is determined by first generating a 64-bit hash of the key.** Thereafter, a mapping to the associated chunk inside the Key-Value window is established:

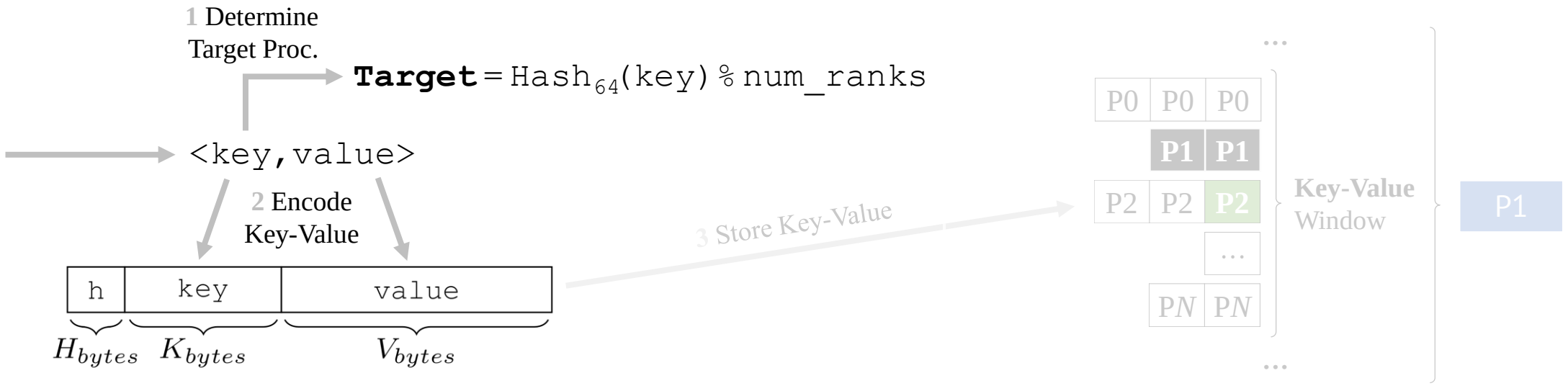
Methodology > Defining the Target Process

Inside the Map phase, **the target process is determined by first generating a 64-bit hash of the key.** Thereafter, a mapping to the associated chunk inside the Key-Value window is established:



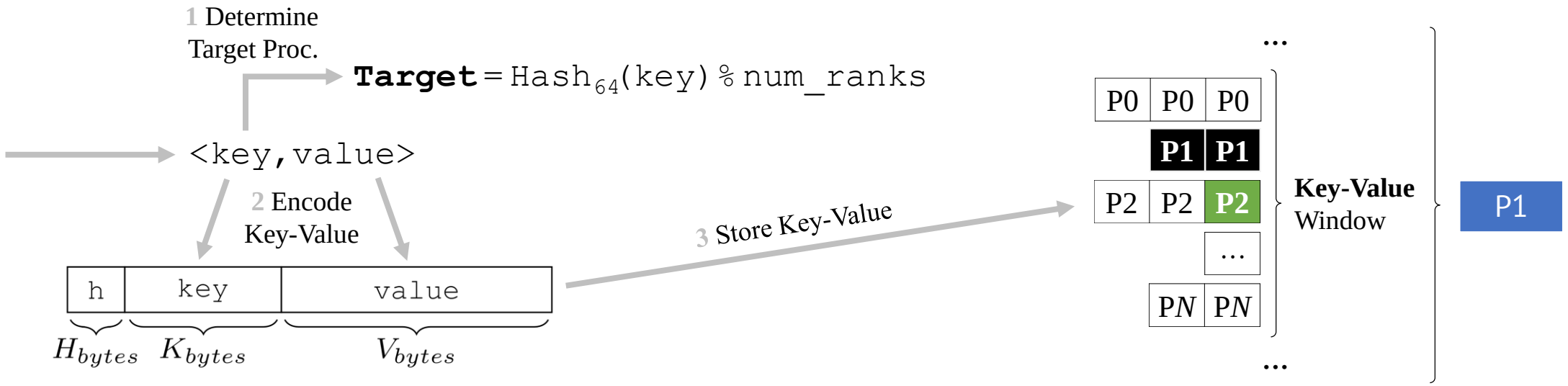
Methodology > Defining the Target Process

Inside the Map phase, **the target process is determined by first generating a 64-bit hash of the key.** Thereafter, a mapping to the associated chunk inside the Key-Value window is established:



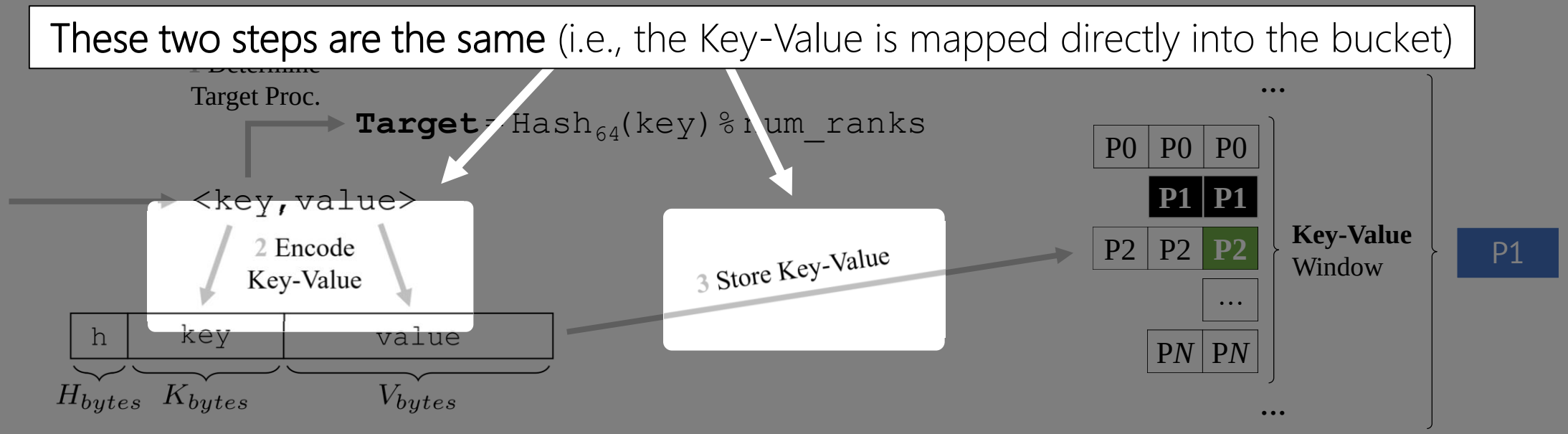
Methodology > Defining the Target Process

Inside the Map phase, **the target process is determined by first generating a 64-bit hash of the key.** Thereafter, a mapping to the associated chunk inside the Key-Value window is established:



Methodology > Defining the Target Process

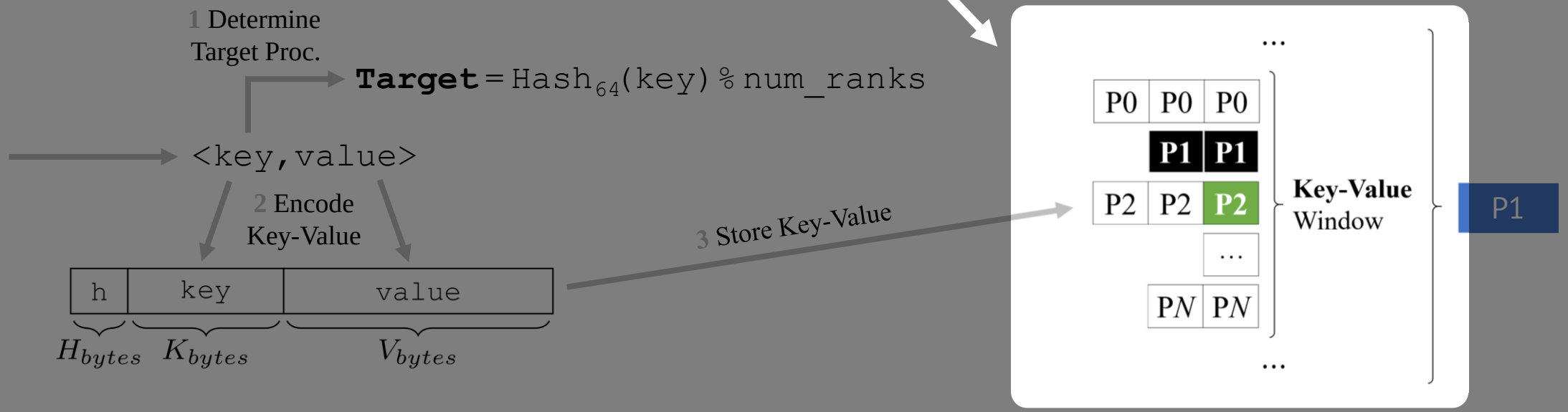
Inside the Map phase, **the target process is determined by first generating a 64-bit hash of the key.** Thereafter, a mapping to the associated chunk inside the Key-Value window is established:



Methodology > Defining the Target Process

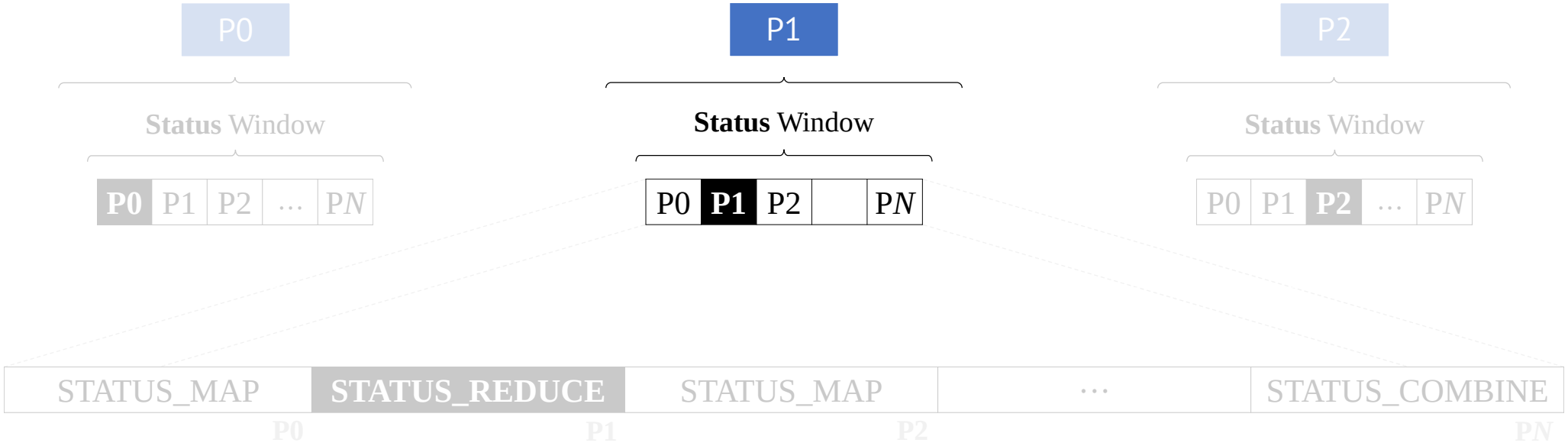
The mechanism allows remote processes to directly reference specific key-values, without affecting surrounding buckets

Thereafter, a



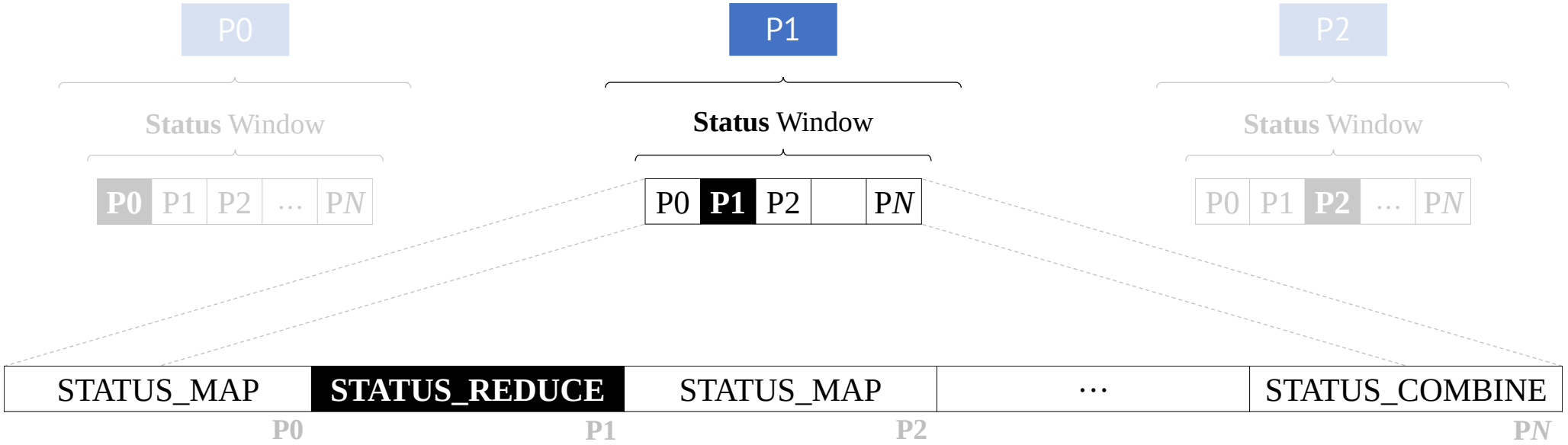
Methodology > Importance of Status Window

The **Status** window is required as a synchronization mechanism to prevent incorrect data accesses to the target **Key-Value** window. Each process has a "process status" array, whose values are updated remotely:



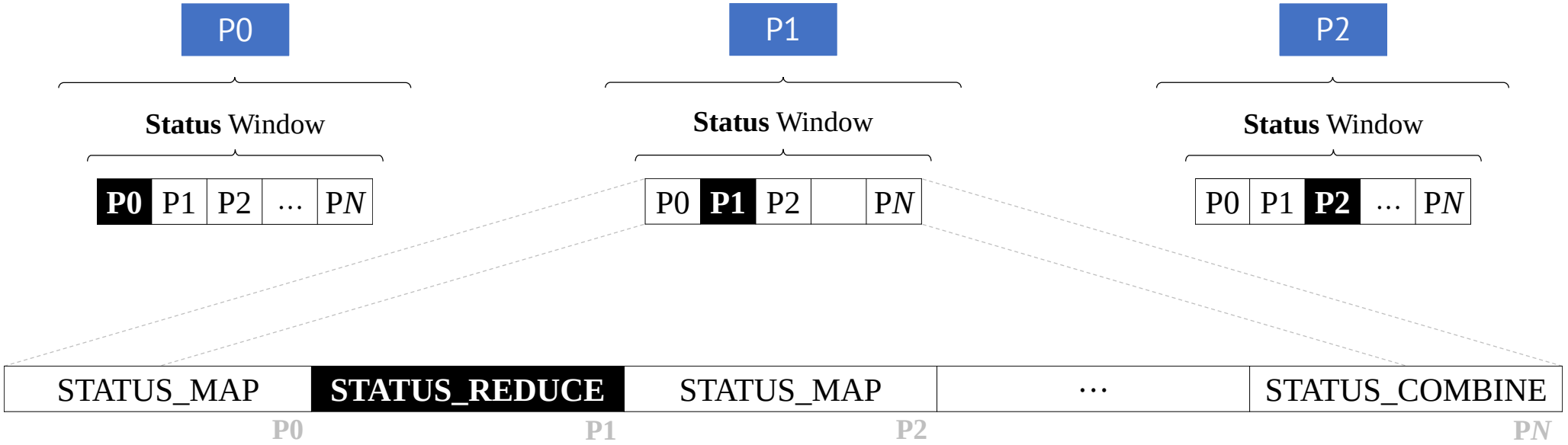
Methodology > Importance of Status Window

The **Status** window is required as a synchronization mechanism to prevent incorrect data accesses to the target **Key-Value** window. Each process has a "process status" array, whose values are updated remotely:



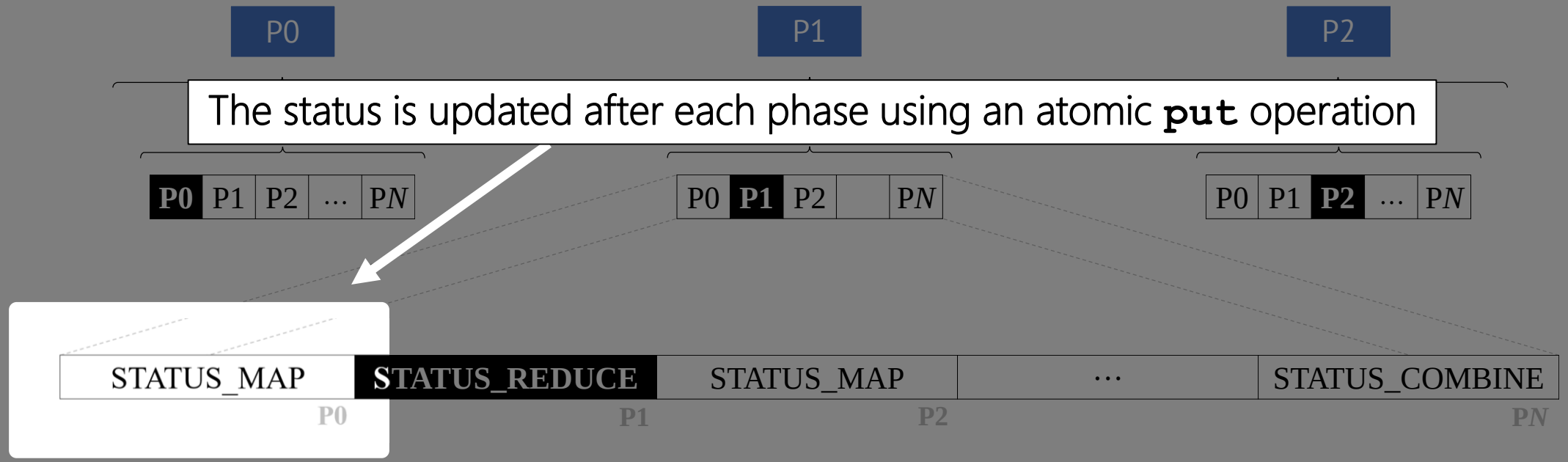
Methodology > Importance of Status Window

The **Status** window is required as a synchronization mechanism to prevent incorrect data accesses to the target **Key-Value** window. Each process has a "process status" array, whose values are updated remotely:



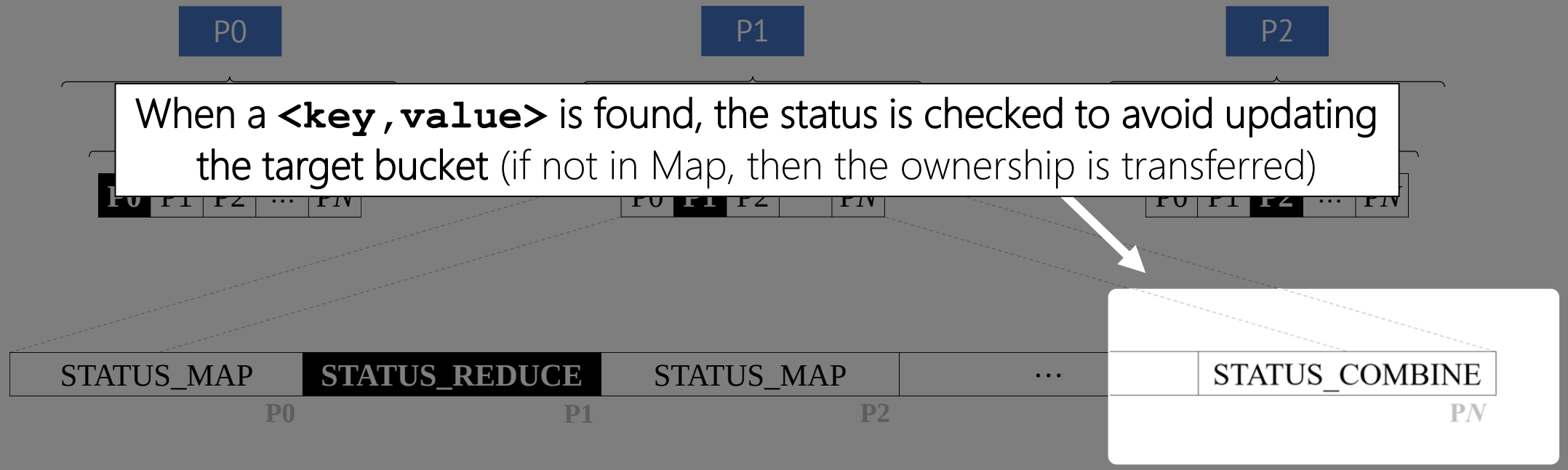
Methodology > Importance of Status Window

The **Status** window is required as a synchronization mechanism to prevent incorrect data accesses to the target **Key-Value** window. Each process has a "process status" array, whose values are updated remotely:



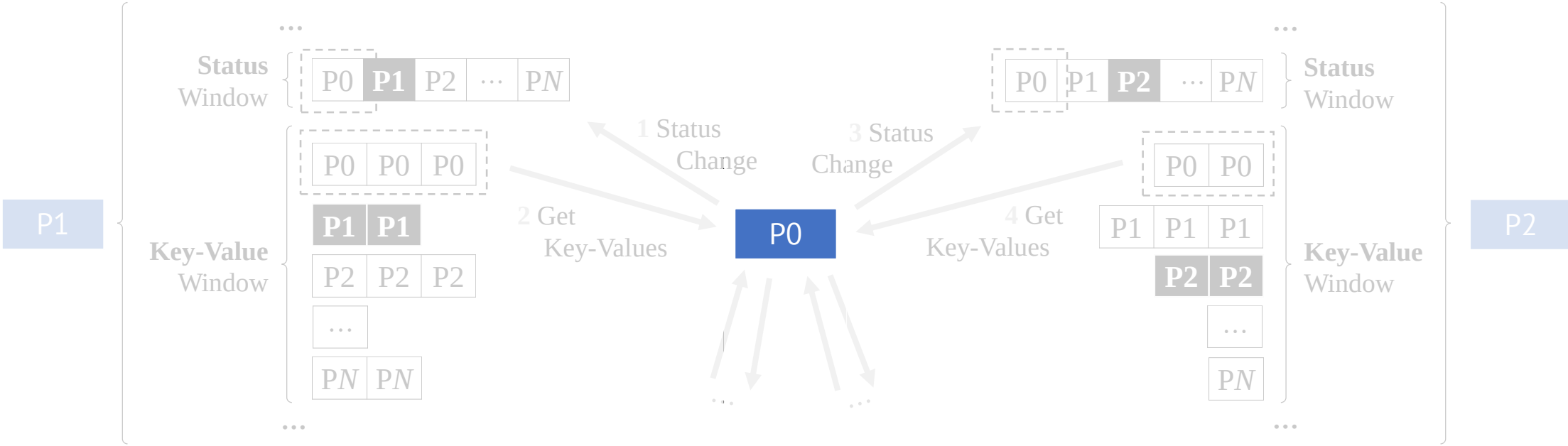
Methodology > Importance of Status Window

The **Status** window is required as a synchronization mechanism to prevent incorrect data accesses to the target **Key-Value** window. Each process has a "process status" array, whose values are updated remotely:



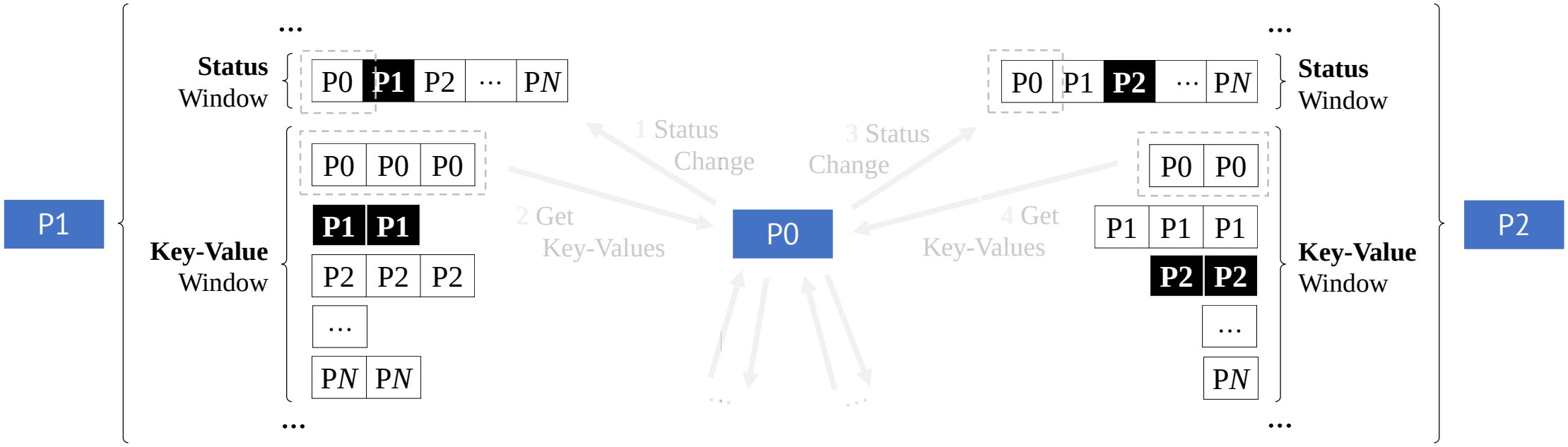
Methodology > Reducing Key-Values

When a process finishes the Map phase, it proceeds to the Reduce phase by **collecting groups of key-value pairs assigned to this particular process, from all the other processes**. The buckets are retrieved using MPI one-sided:



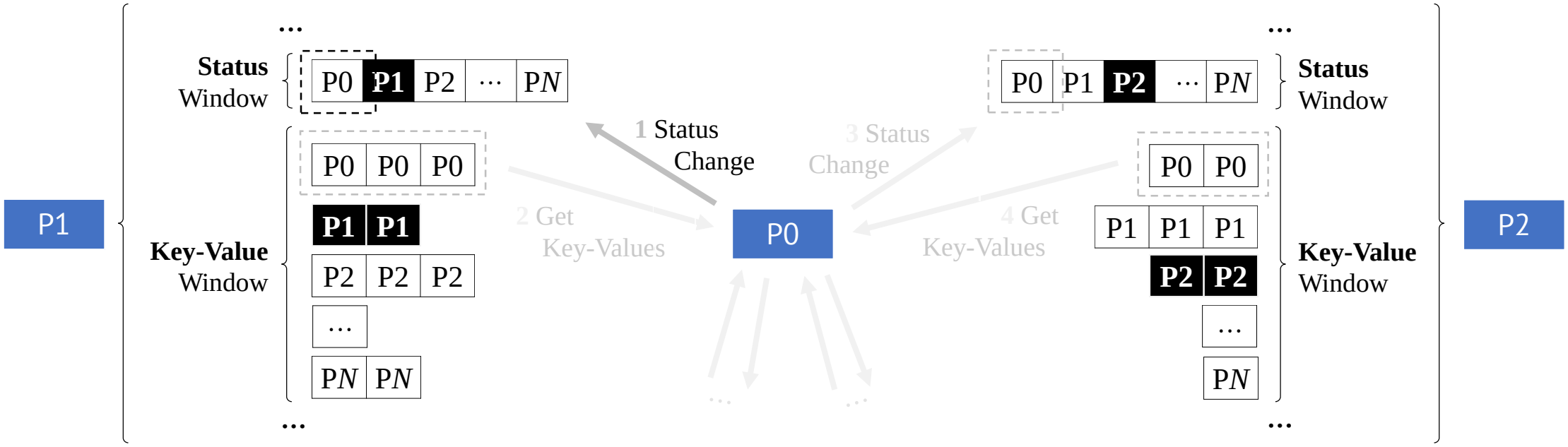
Methodology > Reducing Key-Values

When a process finishes the Map phase, it proceeds to the Reduce phase by **collecting groups of key-value pairs assigned to this particular process, from all the other processes**. The buckets are retrieved using MPI one-sided:



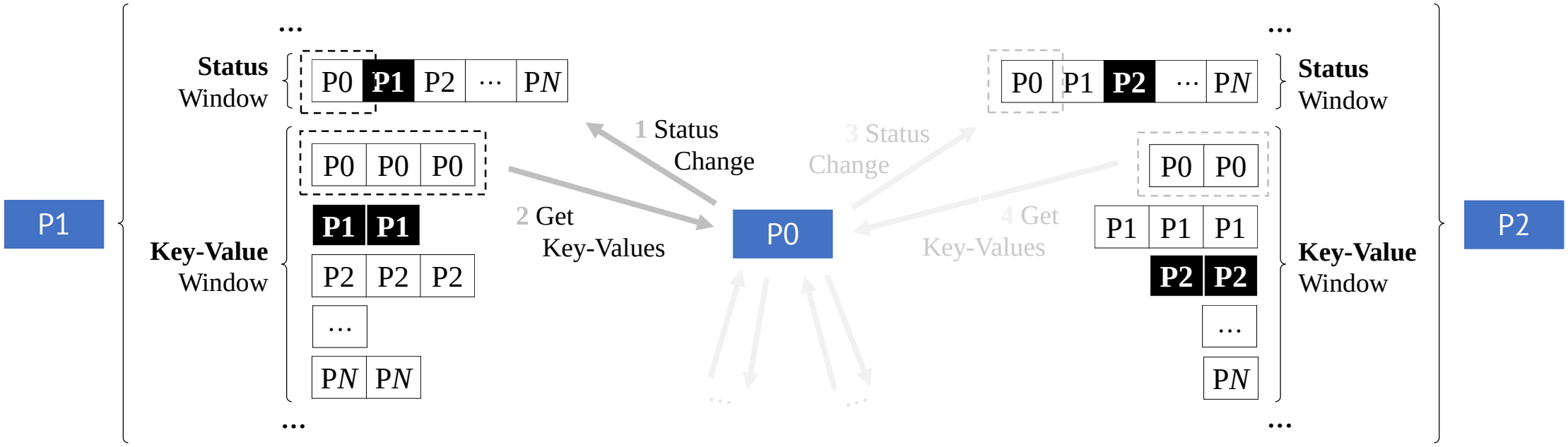
Methodology > Reducing Key-Values

When a process finishes the Map phase, it proceeds to the Reduce phase by **collecting groups of key-value pairs assigned to this particular process, from all the other processes**. The buckets are retrieved using MPI one-sided:



Methodology > Reducing Key-Values

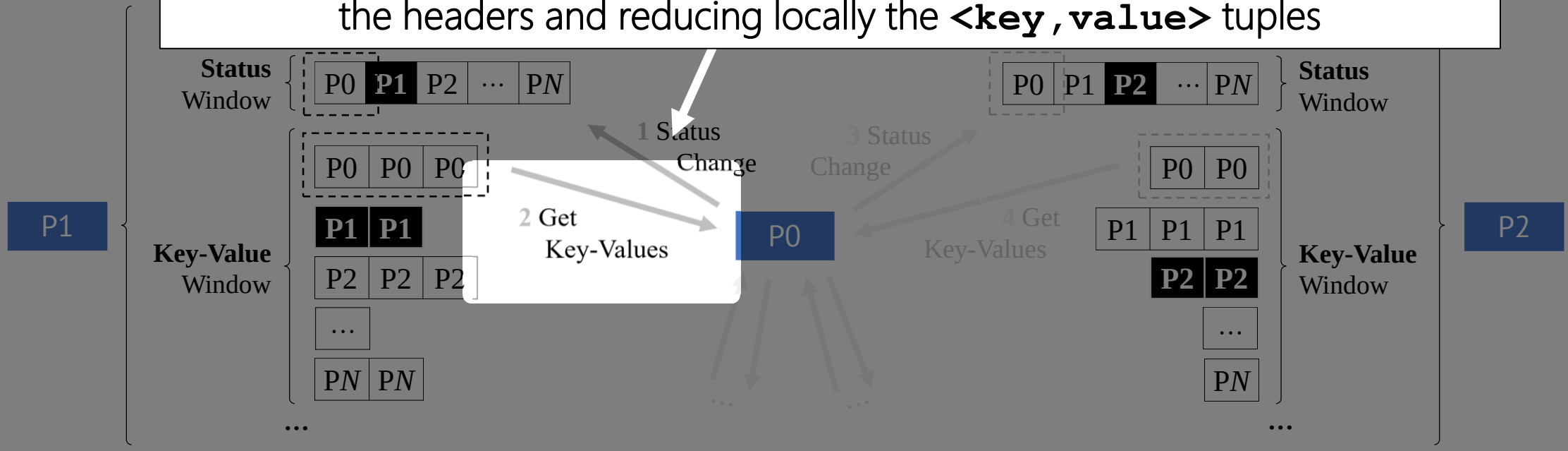
When a process finishes the Map phase, it proceeds to the Reduce phase by **collecting groups of key-value pairs assigned to this particular process, from all the other processes**. The buckets are retrieved using MPI one-sided:



Methodology > Reducing Key-Values

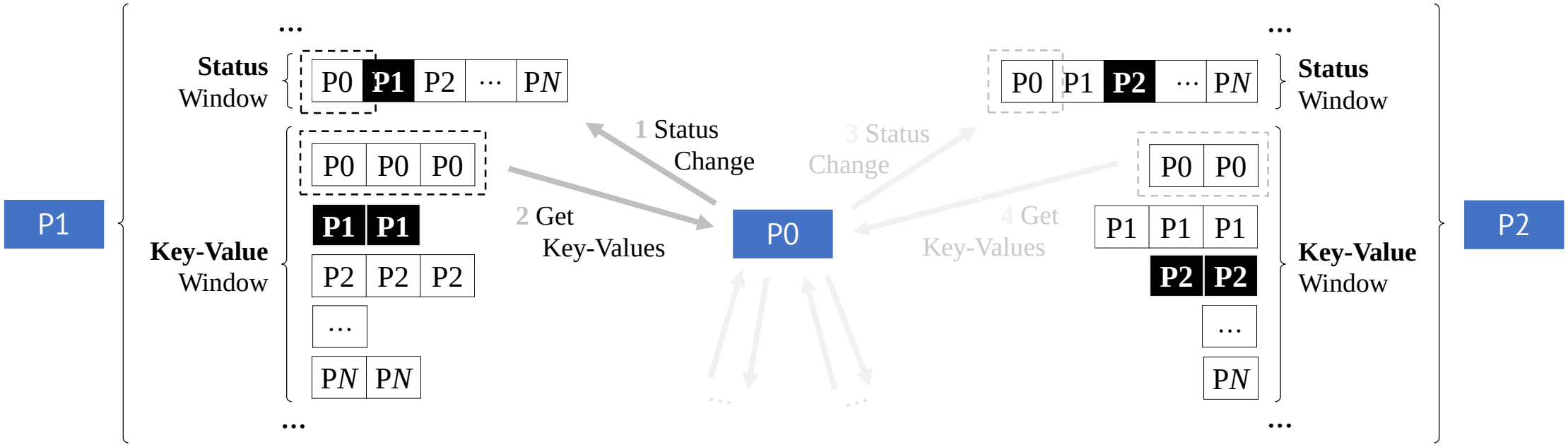
When a process finishes the Map phase, it proceeds to the Reduce phase by **collecting groups of key-value pairs assigned to this particular process, from all the other processes**. The buckets are retrieved using MPI one-sided:

After each bucket is retrieved, the process splits the information by interpreting the headers and reducing locally the **<key,value>** tuples



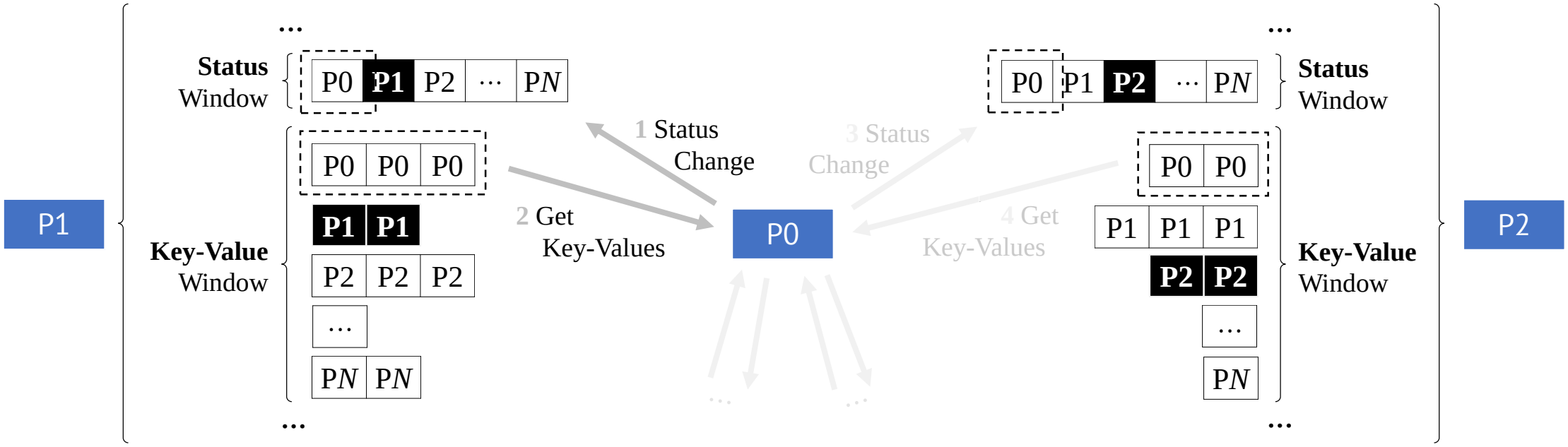
Methodology > Reducing Key-Values

When a process finishes the Map phase, it proceeds to the Reduce phase by **collecting groups of key-value pairs assigned to this particular process, from all the other processes**. The buckets are retrieved using MPI one-sided:



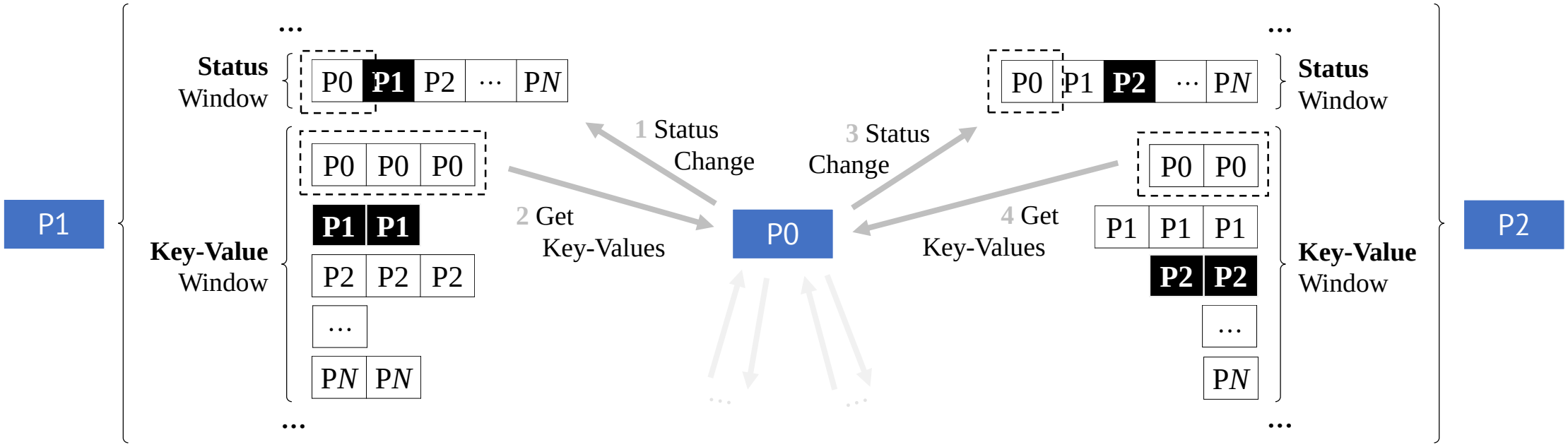
Methodology > Reducing Key-Values

When a process finishes the Map phase, it proceeds to the Reduce phase by **collecting groups of key-value pairs assigned to this particular process, from all the other processes**. The buckets are retrieved using MPI one-sided:



Methodology > Reducing Key-Values

When a process finishes the Map phase, it proceeds to the Reduce phase by **collecting groups of key-value pairs assigned to this particular process, from all the other processes**. The buckets are retrieved using MPI one-sided:

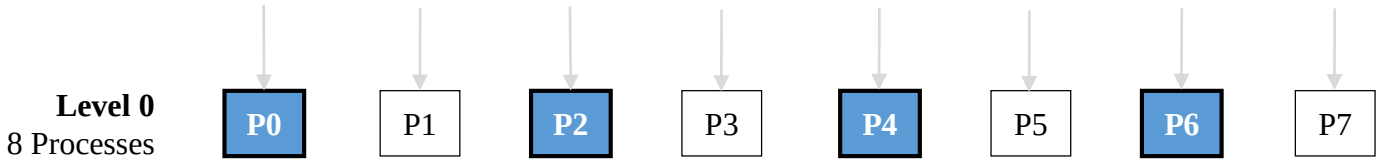


Methodology > Generating the Result

After Map and Reduce are completed, **Combine sets up a tree-based sorting algorithm that fetches key-values from each process to generate the result**. The algorithm used is inspired by *merge sort*:

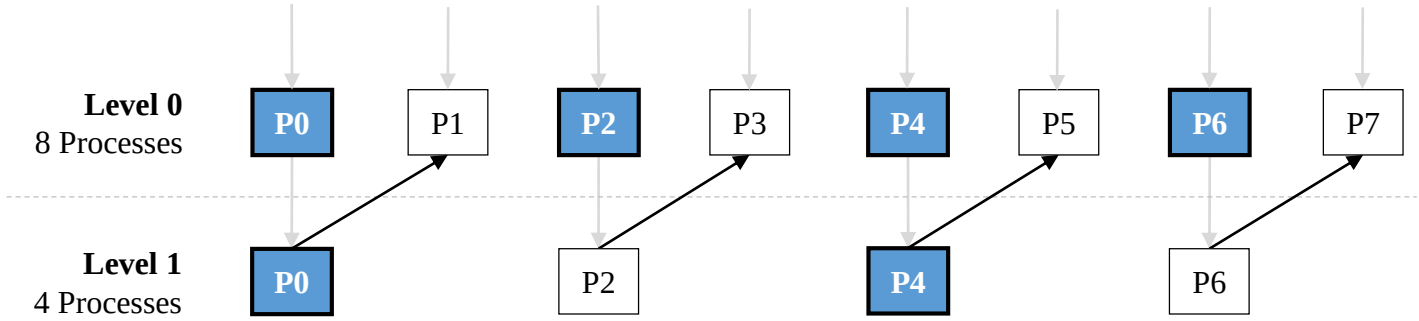
Methodology > Generating the Result

The number of levels in the tree is given by $\lceil \log_2(num\_ranks) \rceil + 1$. **The initial level is common to all the processes and the purpose is to store the local key-value pairs in-order**:



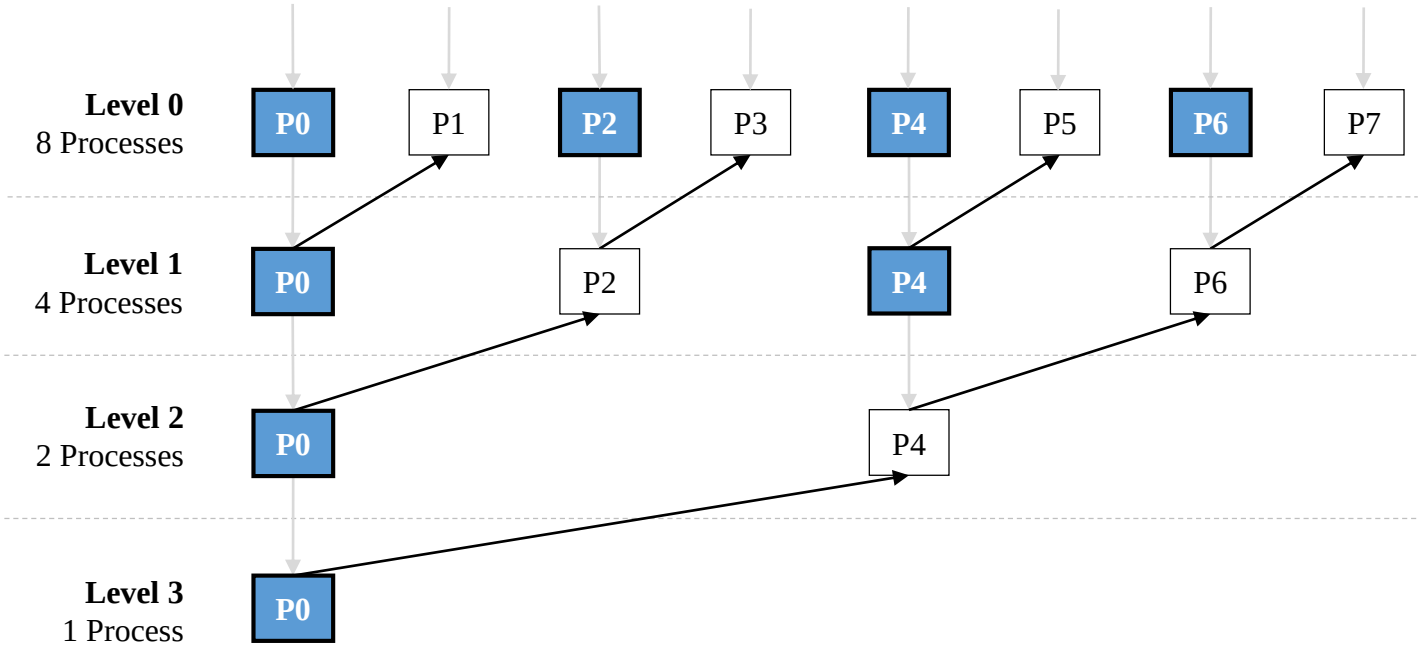
Methodology > Generating the Result

After the initial first step, **the processes retrieve the remote key-values from the previous level using one-sided operations**. This will generate a new level with all the key-value pairs ordered:



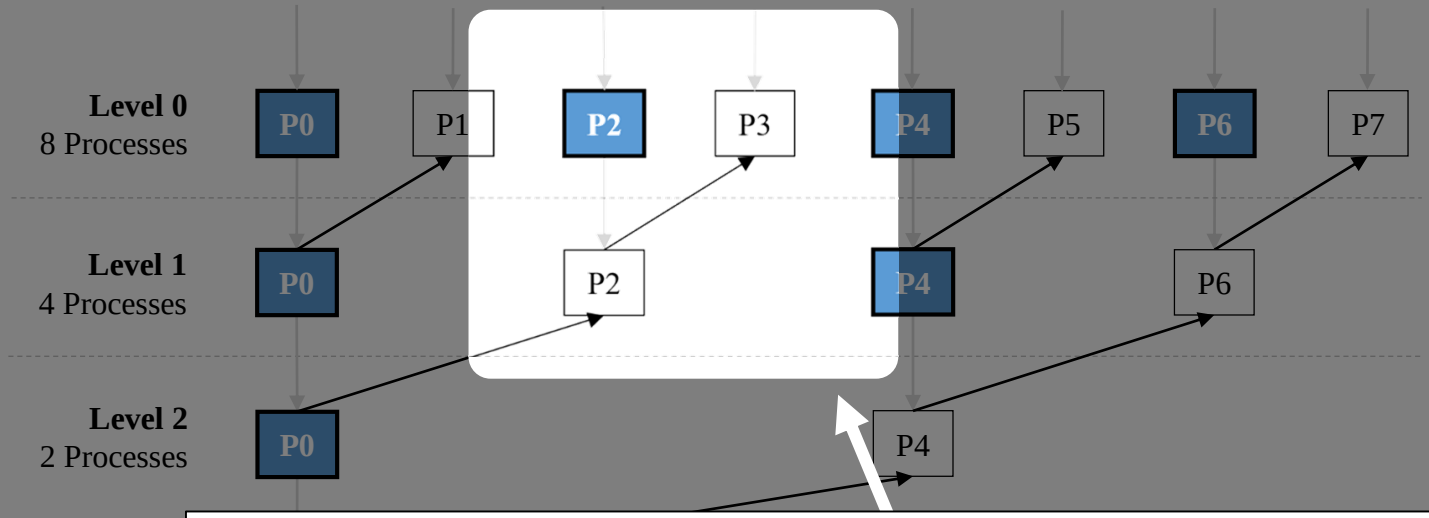
Methodology > Generating the Result

The same task is repeated until one last process generates the final level of the tree, which corresponds to the result of the MapReduce execution. At this point, the algorithm has finished:



Methodology > Generating the Result

The same task is repeated until one last process generates the final level of the tree, which corresponds to the result of the MapReduce execution. At this point, the algorithm has finished:



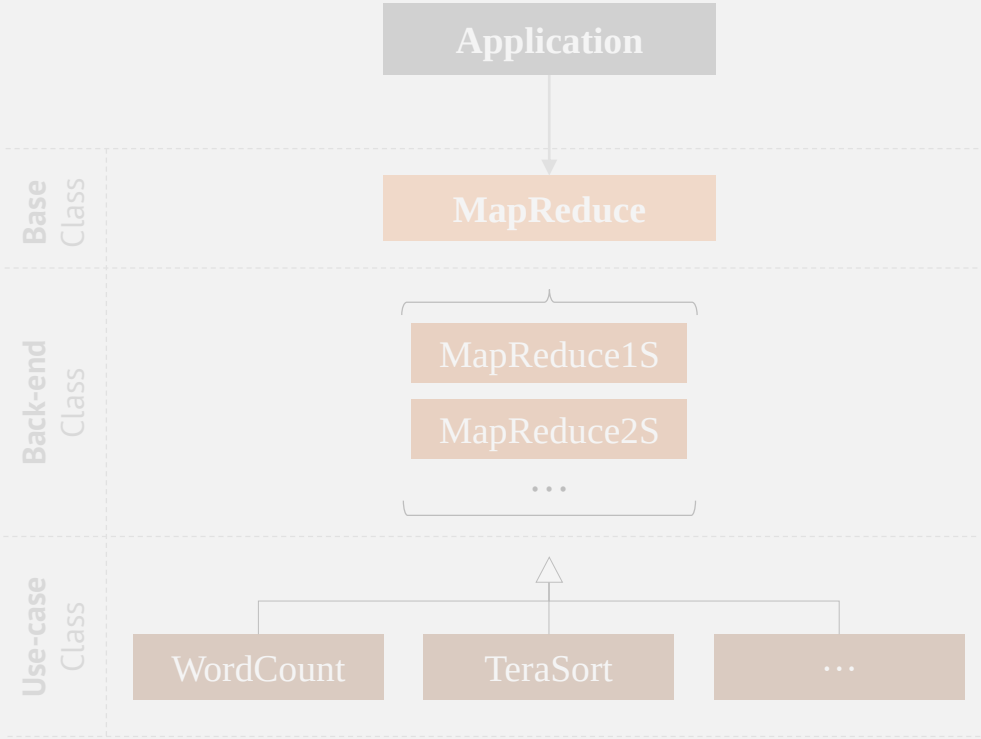
To prevent race conditions over the different levels of the tree, we use an *exclusive* lock (i.e., **`MPI_LOCK_EXCLUSIVE`**) over the Combine window

Methodology > Framework

In order to support our experiments, **we integrate MapReduce-1S as the back-end of a custom MapReduce framework**. The implementation is written in C/C++.

The framework employs a multi-inheritance mechanism by dividing the responsibilities as a hierarchy of classes:

- **Base Class**. Defines the main API to interact with the user, such as initialization or job execution.
- **Back-end Class**. Contains the implementation that performs the phases of the algorithm.
- **Use-case Class**. Exposes the specific `Map()` and `Reduce()` functions required for MapReduce.

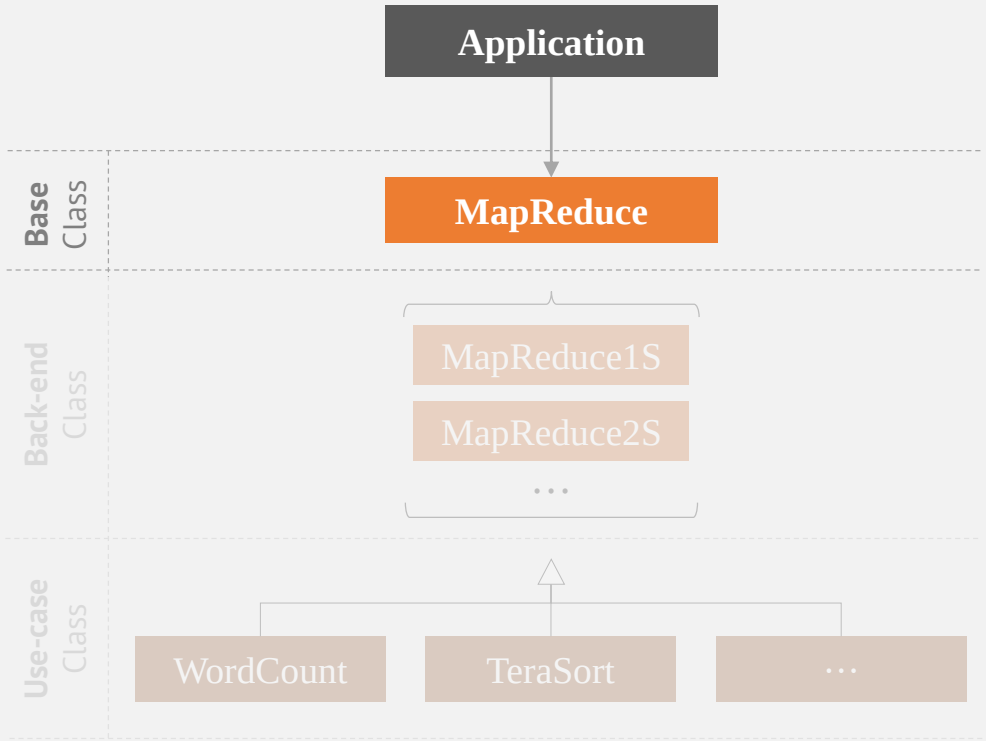


Methodology > Framework

In order to support our experiments, **we integrate MapReduce-1S as the back-end of a custom MapReduce framework**. The implementation is written in C/C++.

The framework employs a multi-inheritance mechanism by dividing the responsibilities as a hierarchy of classes**:**

- **Base Class**. Defines the main API to interact with the user, such as initialization or job execution.
- **Back-end Class**. Contains the implementation that performs the phases of the algorithm.
- **Use-case Class**. Exposes the specific `Map()` and `Reduce()` functions required for MapReduce.

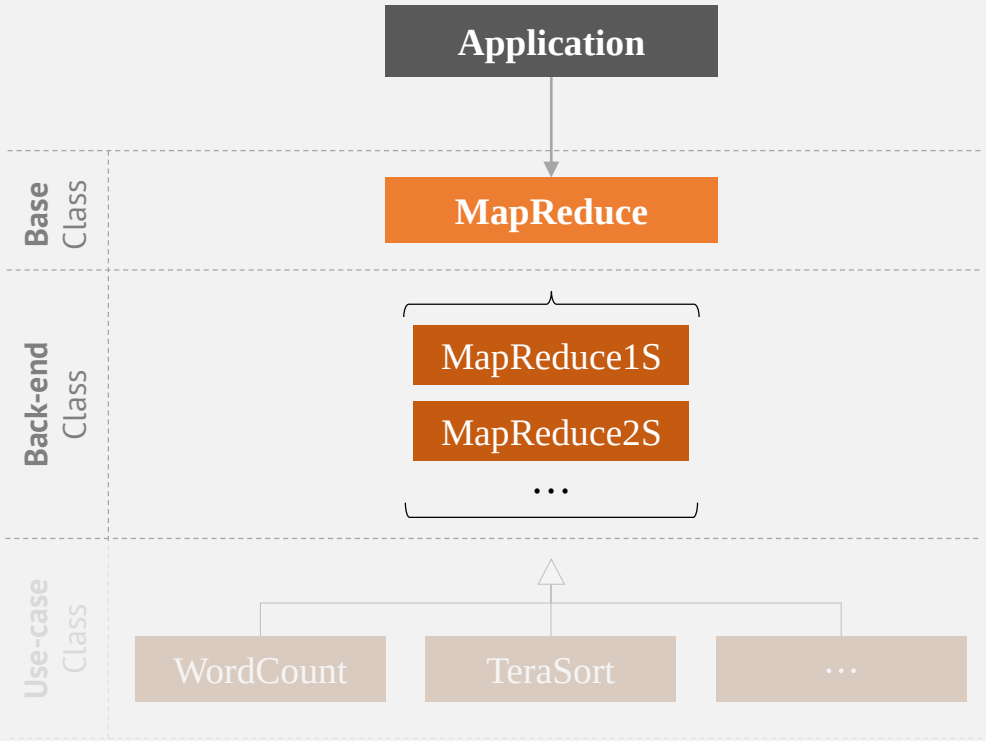


Methodology > Framework

In order to support our experiments, **we integrate MapReduce-1S as the back-end of a custom MapReduce framework**. The implementation is written in C/C++.

The framework employs a multi-inheritance mechanism by dividing the responsibilities as a hierarchy of classes**:**

- **Base Class**. Defines the main API to interact with the user, such as initialization or job execution.
- **Back-end Class**. Contains the implementation that performs the phases of the algorithm.
- **Use-case Class**. Exposes the specific `Map()` and `Reduce()` functions required for MapReduce.



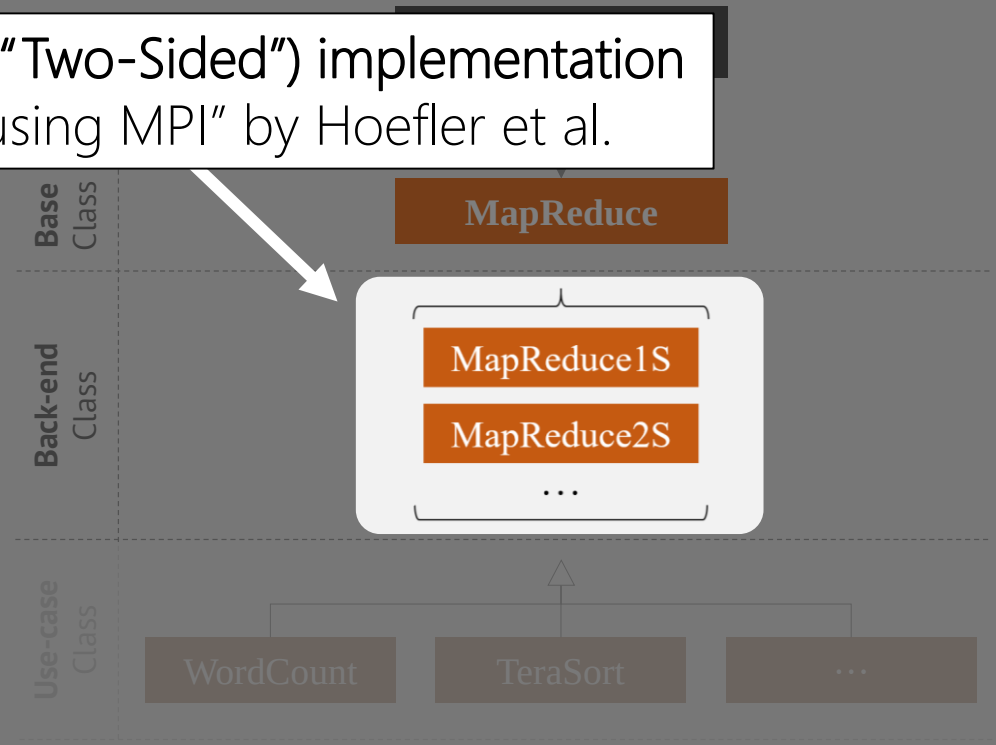
Methodology > Framework

We also integrate a MapReduce-2S (i.e., MapReduce "Two-Sided") implementation based on the work "Towards efficient MapReduce using MPI" by Hoefler et al.

In
Ma
framework. The implementation is written in C/C++.

The framework employs a multi-inheritance mechanism by dividing the responsibilities as a hierarchy of classes:

- **Base Class**. Defines the main API to interact with the user, such as initialization or job execution.
- **Back-end Class**. Contains the implementation that performs the phases of the algorithm.
- **Use-case Class**. Exposes the specific `Map()` and `Reduce()` functions required for MapReduce.

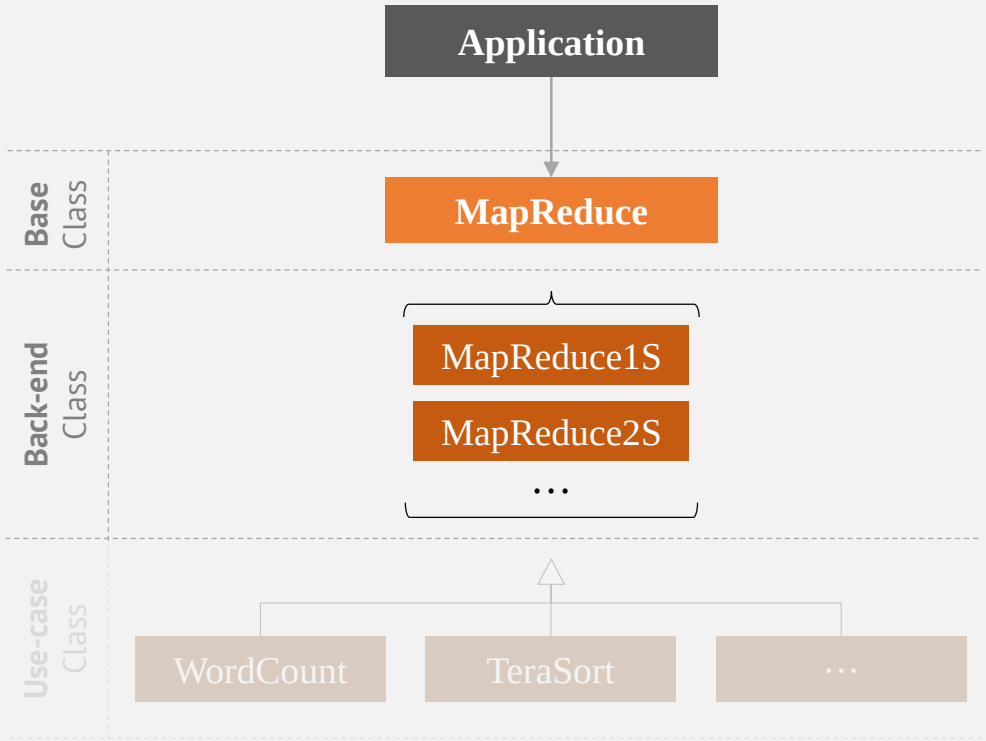


Methodology > Framework

In order to support our experiments, **we integrate MapReduce-1S as the back-end of a custom MapReduce framework**. The implementation is written in C/C++.

The framework employs a multi-inheritance mechanism by dividing the responsibilities as a hierarchy of classes**:**

- **Base Class**. Defines the main API to interact with the user, such as initialization or job execution.
- **Back-end Class**. Contains the implementation that performs the phases of the algorithm.
- **Use-case Class**. Exposes the specific `Map()` and `Reduce()` functions required for MapReduce.

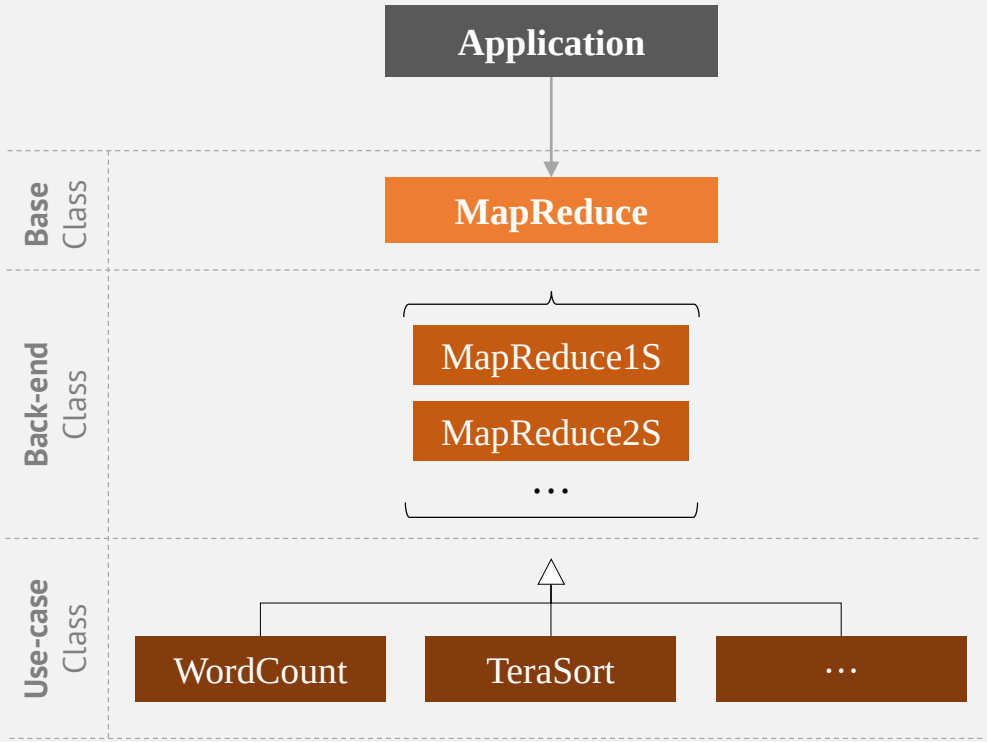


Methodology > Framework

In order to support our experiments, **we integrate MapReduce-1S as the back-end of a custom MapReduce framework**. The implementation is written in C/C++.

The framework employs a multi-inheritance mechanism by dividing the responsibilities as a hierarchy of classes**:**

- **Base Class**. Defines the main API to interact with the user, such as initialization or job execution.
- **Back-end Class**. Contains the implementation that performs the phases of the algorithm.
- **Use-case Class**. Exposes the specific `Map()` and `Reduce()` functions required for MapReduce.



Methodology > Framework

In order to support our experiments, **we integrate MapReduce-1S as the back-end of a custom MapReduce framework**. The implementation is written in C/C++.

The framework employs a multi-inheritance mechanism by dividing the responsibilities as a hierarchy of classes:

- **Base Class**. Defines the main API to interact with the user, such as initialization or job execution.
- **Back-end Class**. Contains the implementation that performs the phases of the algorithm.
- **Use-case Class**. Exposes the specific `Map()` and `Reduce()` functions required for MapReduce.

```
...

// Create the MR object with MR-1S as back-end
MapReduce1S *map_reduce = new WordCount();

// Init the job with the specific settings
map_reduce->Init(input_filename, init_win_size,
                 max_chunk_size, max_task_size,
                 storage_enabled, hash_enabled,
                 input_api, sfactor, sunit);

// Launch the execution and output the result
map_reduce->Run();
map_reduce->Print();

// Close the job and release the object
map_reduce->Finalize();
delete map_reduce;

...
```

Experimental Results

We estimate the overall performance of MapReduce-1S compared to MapReduce-2S. We aim to understand how our approach could fit in existing MapReduce frameworks:

Specifications of Supercomputer "Tegner"	
Nodes	46 × Haswell-based compute nodes
Processor	2 × 12-core Haswell E5-2690v3 @ 2.6GHz
Memory	512GB DRAM per node
Storage	5PB Lustre PFS (Client v2.5.2) with 165 × OST Servers
Software	CentOS v7.4.1708 / Intel ICC and Intel MPI v18.0.1

All the figures reflect the standard deviation of the samples as error bars. **The terms "MR-1S" and "MR-2S" are used to refer to MapReduce-1S and MapReduce-2S**.

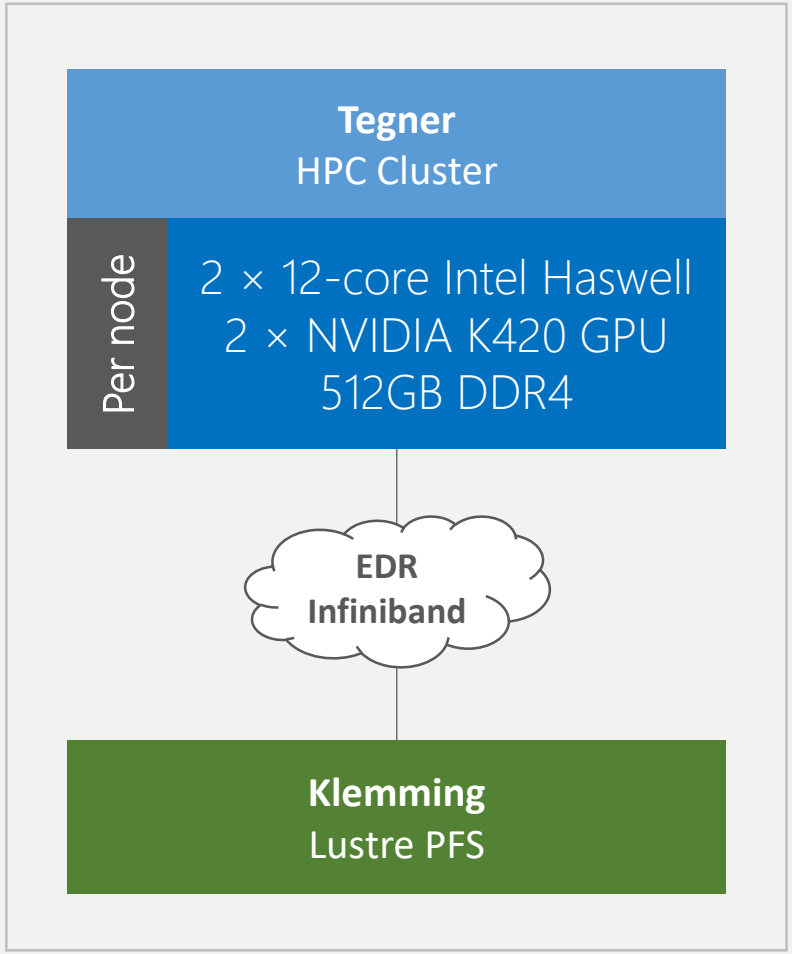
HPC System for Visualization and Big Data

Experimental Results

We estimate the overall performance of MapReduce-1S compared to MapReduce-2S. We aim to understand how our approach could fit in existing MapReduce frameworks:

Specifications of Supercomputer "Tegner"	
Nodes	46 × Haswell-based compute nodes
Processor	2 × 12-core Haswell E5-2690v3 @ 2.6GHz
Memory	512GB DRAM per node
Storage	5PB Lustre PFS (Client v2.5.2) with 165 × OST Servers
Software	CentOS v7.4.1708 / Intel ICC and Intel MPI v18.0.1

All the figures reflect the standard deviation of the samples as error bars. **The terms "MR-1S" and "MR-2S" are used to refer to MapReduce-1S and MapReduce-2S**.



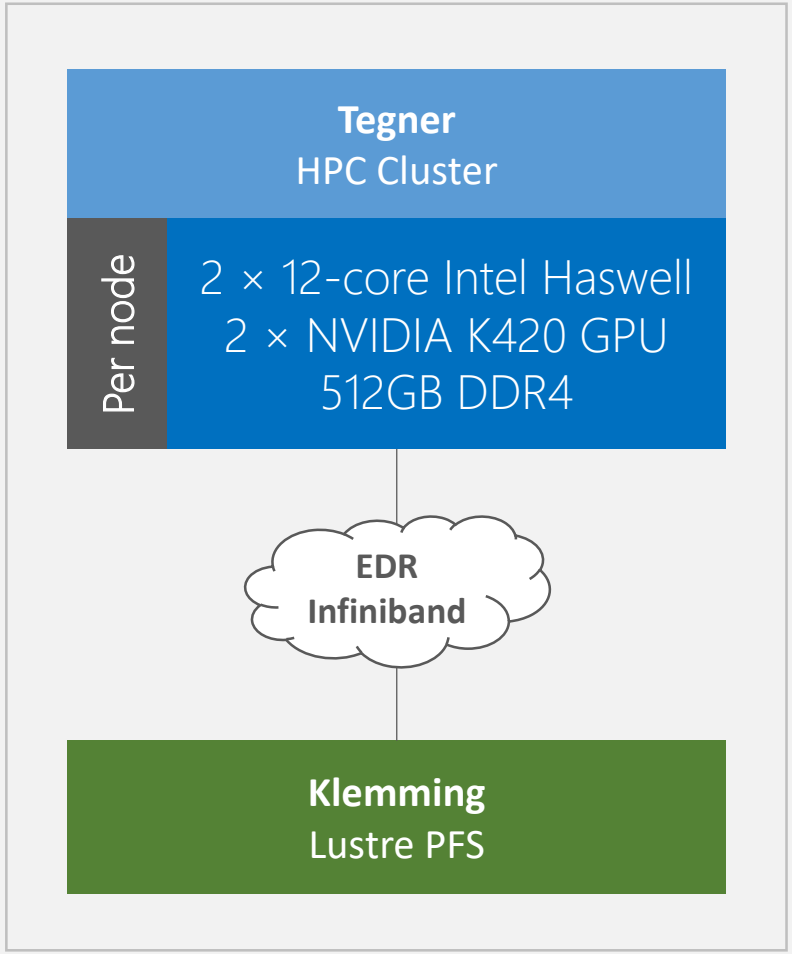
HPC System for Visualization and Big Data

Experimental Results

We estimate the overall performance of MapReduce-1S compared to MapReduce-2S. We aim to understand how our approach could fit in existing MapReduce frameworks:

Specifications of Supercomputer "Tegner"	
Nodes	46 × Haswell-based compute nodes
Processor	2 × 12-core Haswell E5-2690v3 @ 2.6GHz
Memory	512GB DRAM per node
Storage	5PB Lustre PFS (Client v2.5.2) with 165 × OST Servers
Software	CentOS v7.4.1708 / Intel ICC and Intel MPI v18.0.1

All the figures reflect the standard deviation of the samples as error bars. **The terms "MR-1S" and "MR-2S" are used to refer to MapReduce-1S and MapReduce-2S**.



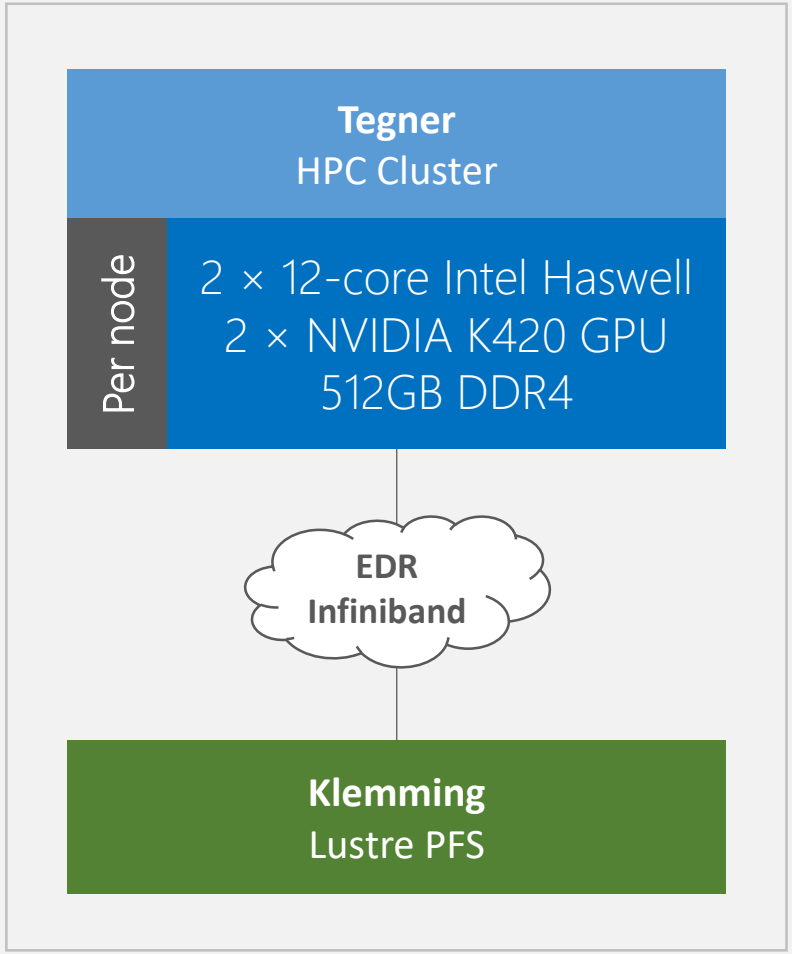
HPC System for Visualization and Big Data

Experimental Results

We estimate the overall performance of MapReduce-1S compared to MapReduce-2S. We aim to understand how our approach could fit in existing MapReduce frameworks:

Specifications of Supercomputer "Tegner"	
Nodes	46 × Haswell-based compute nodes
Processor	2 × 12-core Haswell E5-2690v3 @ 2.6GHz
Memory	512GB DRAM per node
Storage	5PB Lustre PFS (Client v2.5.2) with 165 × OST Servers
Software	CentOS v7.4.1708 / Intel ICC and Intel MPI v18.0.1

All the figures reflect the standard deviation of the samples as error bars. **The terms "MR-1S" and "MR-2S" are used to refer to MapReduce-1S and MapReduce-2S**.



HPC System for Visualization and Big Data

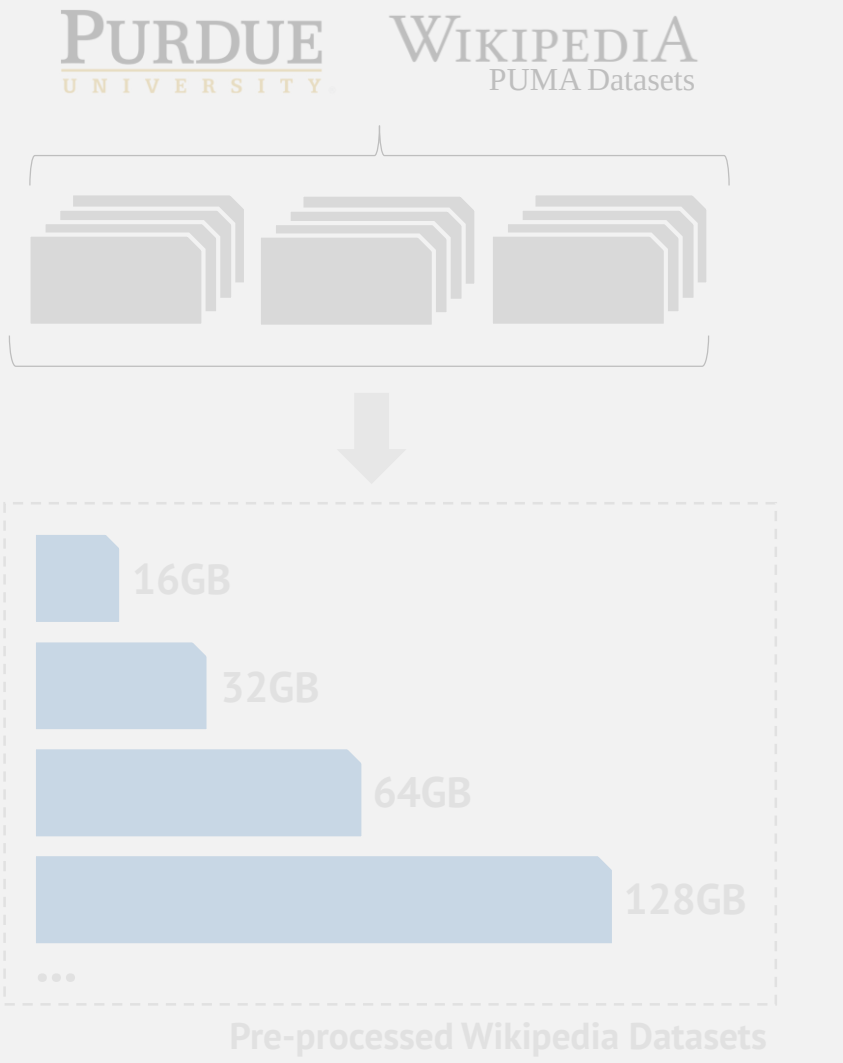
Experimental Results > Word-Count

We evaluate the scalability of MapReduce-1S using Word-Count, a technique that has major relevance in Big Data analytics.

The basic principle of Word-Count is to compute the occurrences of individual words over large collections of documents:

- The Map phase emits `<word,1>` key-value pairs, where `word` represents the key and `1` the occurrence found.
- The Reduce phase aggregates the occurrences for a given `word` to generate its final `<word,count>`.
- Finally, the Combine phase aggregates all the key-values to produce the result.

We use the PUMA-Wikipedia datasets from PURDUE University
https://engineering.purdue.edu/~puma/datasets.htm



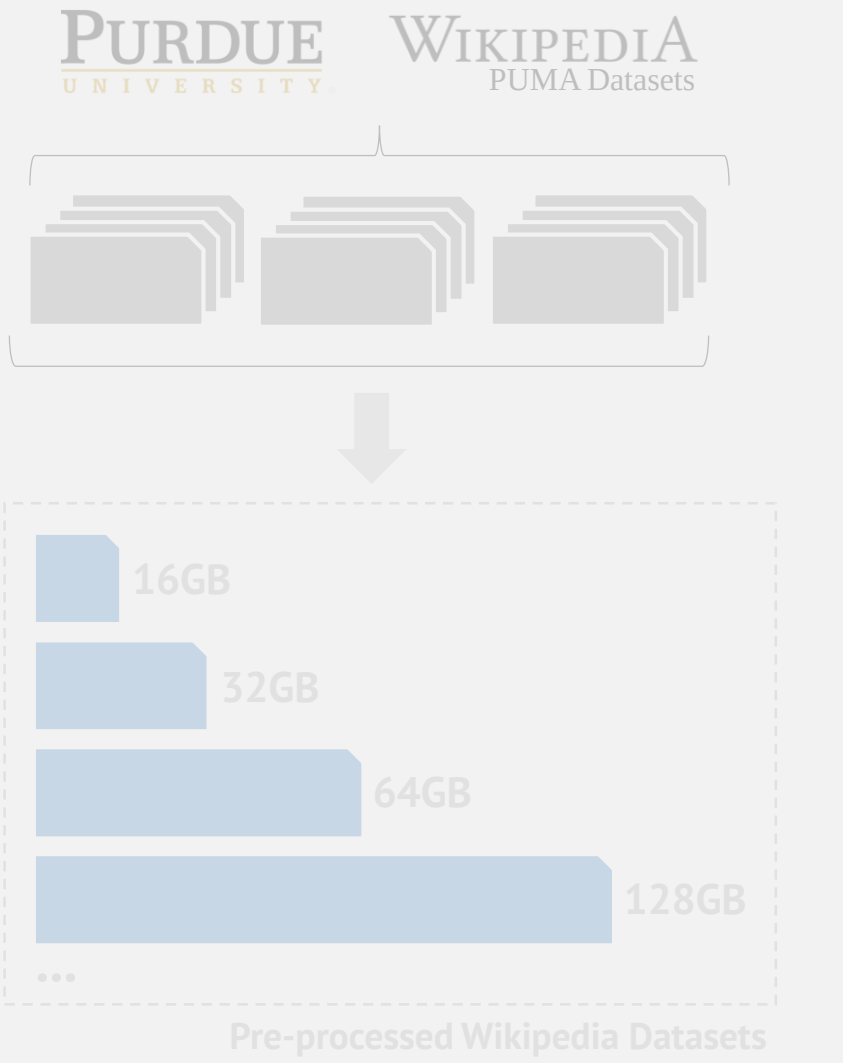
Experimental Results > Word-Count

We evaluate the scalability of MapReduce-1S using Word-Count, a technique that has major relevance in Big Data analytics.

The basic principle of Word-Count is to compute the occurrences of individual words over large collections of documents:

- The Map phase emits `<word,1>` key-value pairs, where `word` represents the key and `1` the occurrence found.
- The Reduce phase aggregates the occurrences for a given `word` to generate its final `<word,count>`.
- Finally, the Combine phase aggregates all the key-values to produce the result.

We use the PUMA-Wikipedia datasets from PURDUE University
https://engineering.purdue.edu/~puma/datasets.htm



Experimental Results > Word-Count

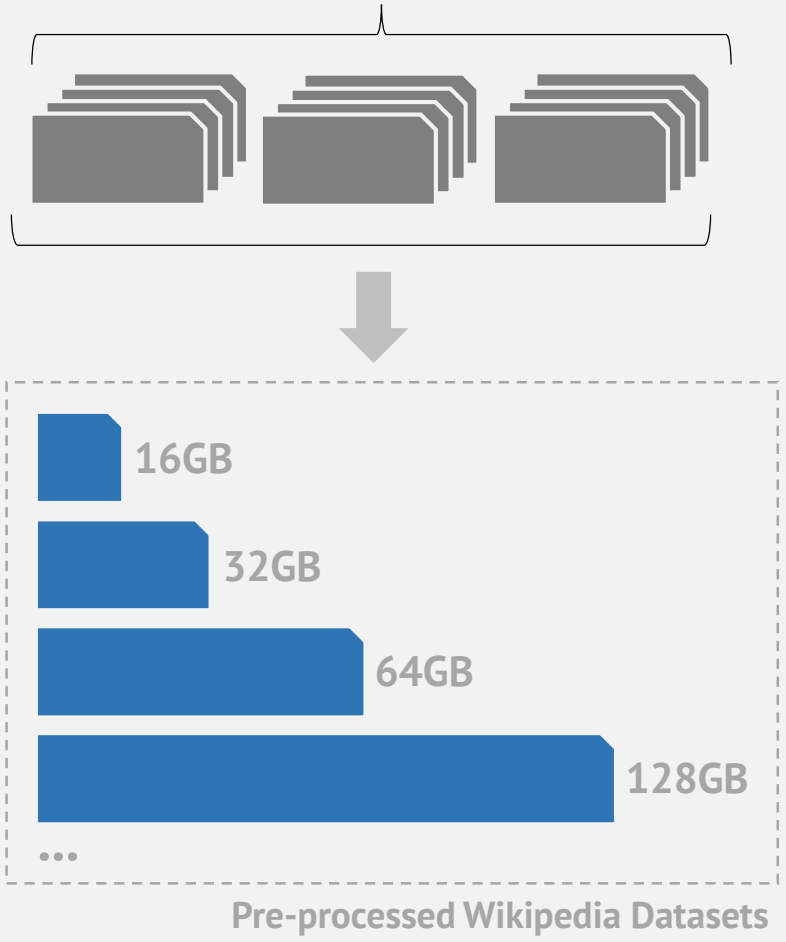
We evaluate the scalability of MapReduce-1S using Word-Count, a technique that has major relevance in Big Data analytics.

The basic principle of Word-Count is to compute the occurrences of individual words over large collections of documents:

- The Map phase emits `<word,1>` key-value pairs, where `word` represents the key and `1` the occurrence found.
- The Reduce phase aggregates the occurrences for a given `word` to generate its final `<word,count>`.
- Finally, the Combine phase aggregates all the key-values to produce the result.

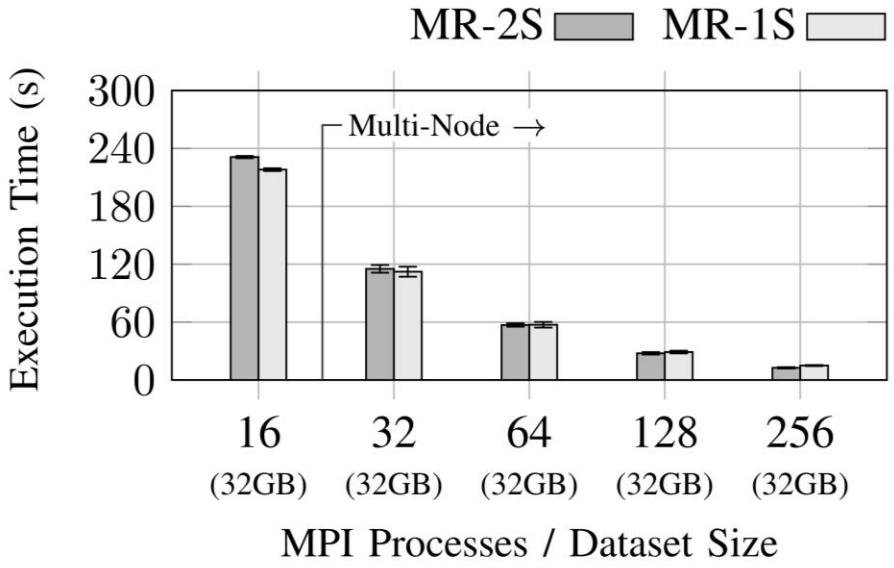
We use the PUMA-Wikipedia datasets from PURDUE University
https://engineering.purdue.edu/~puma/datasets.htm

PURDUE UNIVERSITY WIKIPEDIA PUMA Datasets



Experimental Results > Performance Evaluation

The following figure illustrates the performance of each approach by varying the number of MPI processes on Tegner for a fixed-size and variable-size input datasets (i.e., strong / weak scaling), **under balanced workload**:



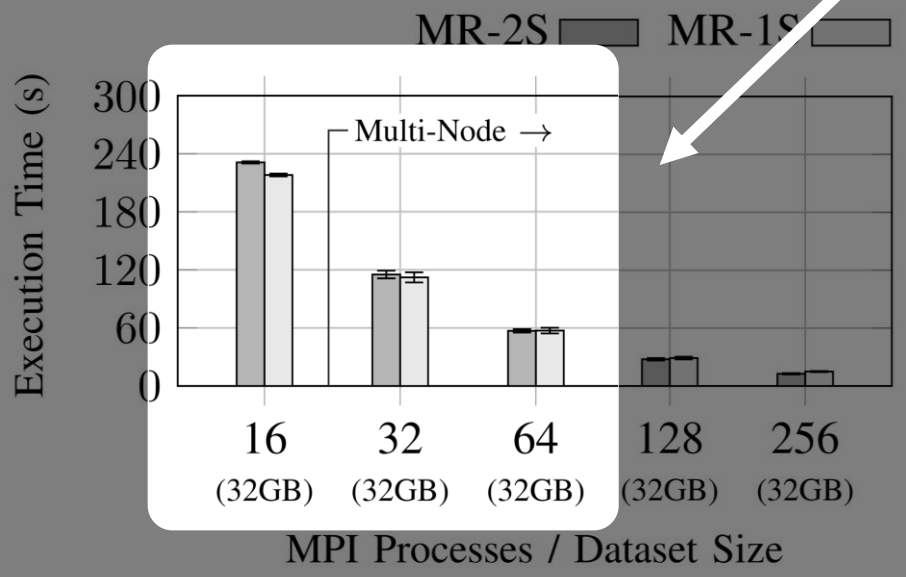
(a) **Strong Scaling Evaluation** under Balanced Workload

(b) **Weak Scaling Evaluation** under Balanced Workload

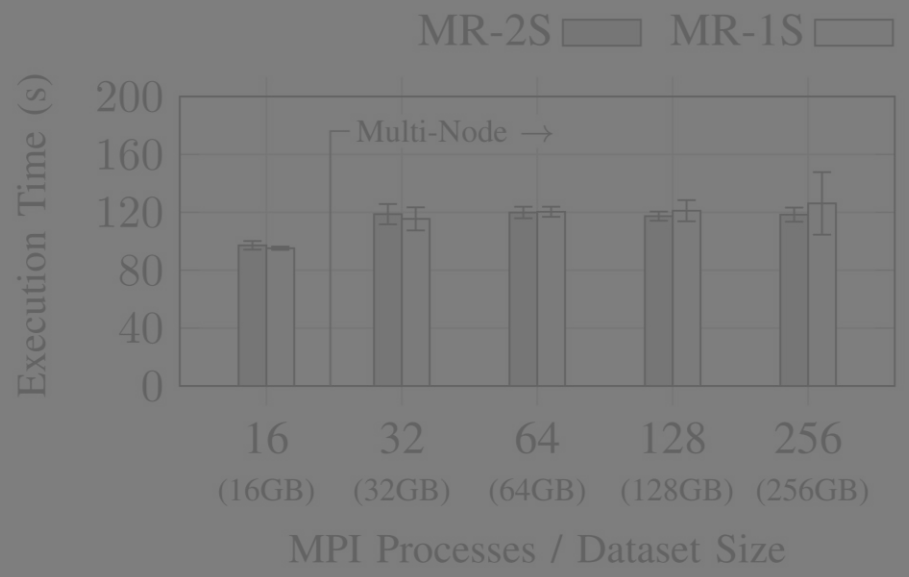
Experimental Results > Performance Evaluation

The following figure illustr processes on Tegner for a fixed-size and rkload:

MapReduce-1S provides approximately 4.8% improvement on average for lower process counts (up to 64 processes)



(a) **Strong Scaling Evaluation** under Balanced Workload

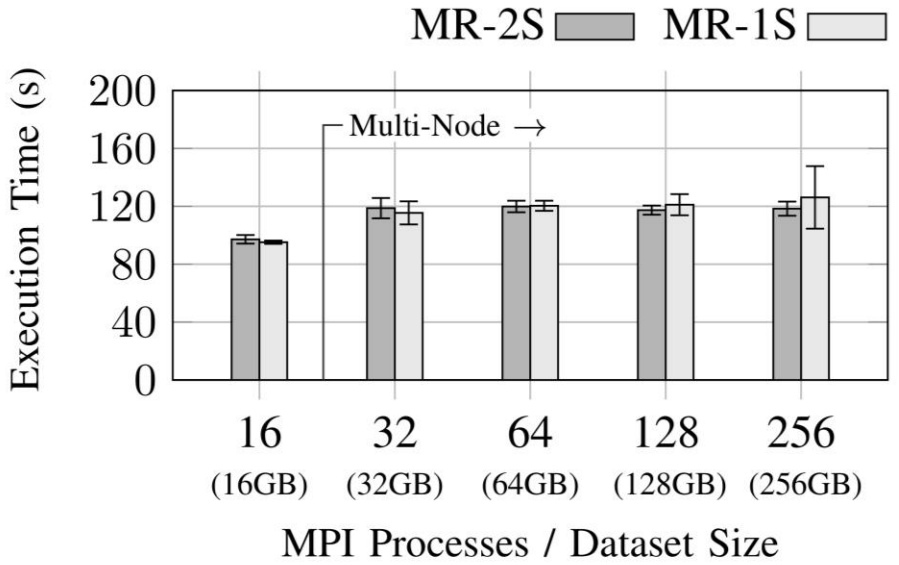


(b) **Weak Scaling Evaluation** under Balanced Workload

Experimental Results > Performance Evaluation

The following figure illustrates the performance of each approach by varying the number of MPI processes on Tegner for a fixed-size and variable-size input datasets (i.e., strong / weak scaling), **under balanced workload**:

(a) **Strong Scaling Evaluation** under Balanced Workload

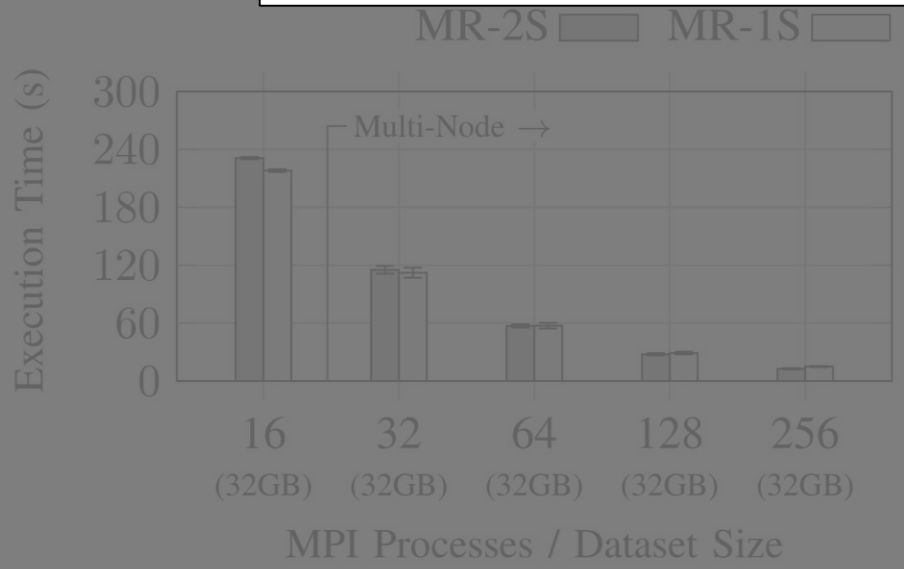


(b) **Weak Scaling Evaluation** under Balanced Workload

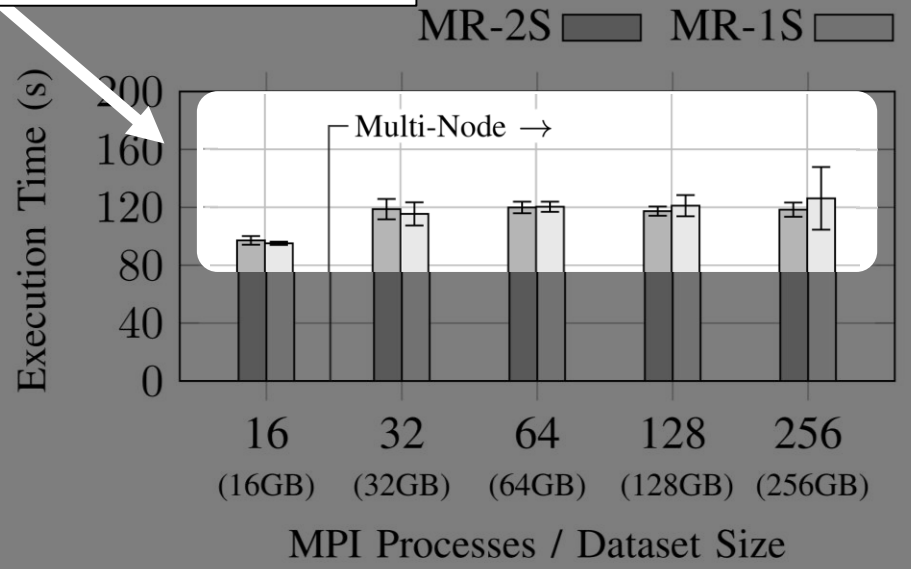
Experimental Results > Performance Evaluation

The following figure illustrates the performance of each approach by varying the number of MPI processes on Tegner for a fixe **der balanced workload**:

The average execution time is 111.3 seconds for MapReduce-2S, and 111.8 seconds for MapReduce-1S (0.5% difference)



(a) **Strong Scaling Evaluation** under Balanced Workload

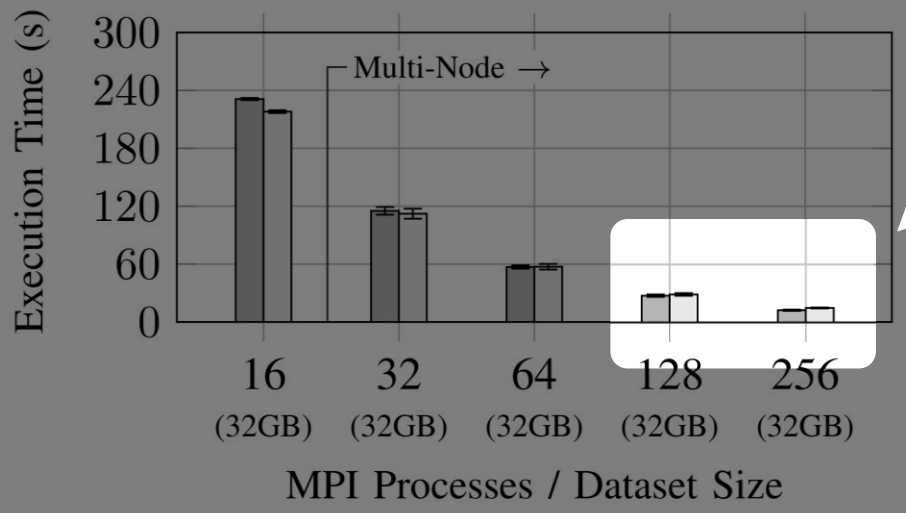


(b) **Weak Scaling Evaluation** under Balanced Workload

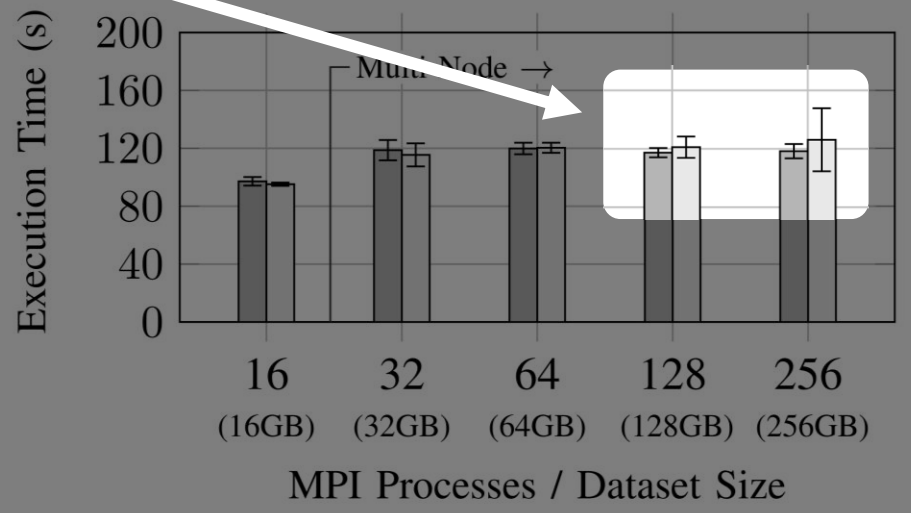
Experimental Results > Performance Evaluation

The following figure illustrates the performance of each approach by varying the number of MPI processes on Tegner for a fix

The performance of MapReduce-1S is clearly affected as the process counts increases
Hence, collective communication and collective I/O results in better performance



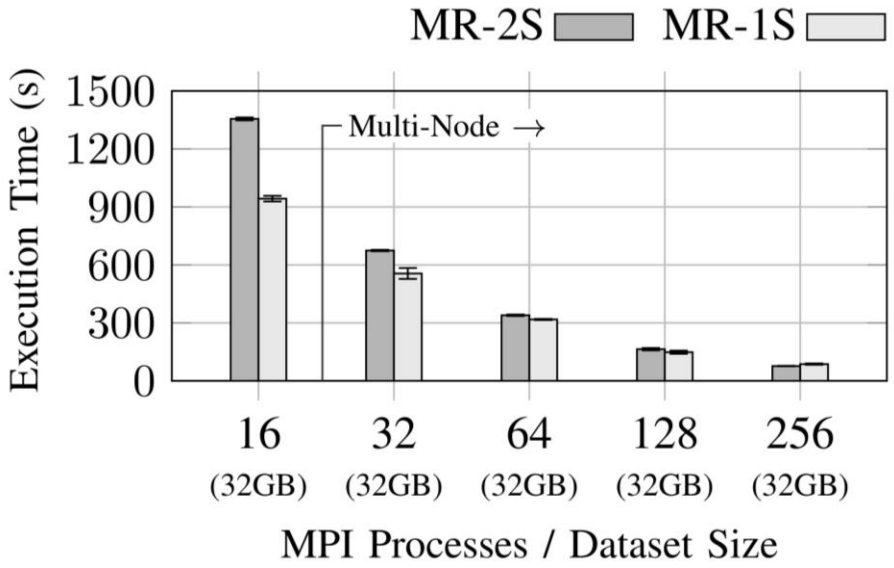
(a) **Strong Scaling Evaluation** under Balanced Workload



(b) **Weak Scaling Evaluation** under Balanced Workload

Experimental Results > Performance Evaluation

The following figure illustrates the performance of each approach by varying the number of MPI processes on Tegner for a fixed-size and variable-size input datasets (i.e., strong / weak scaling), **under unbalanced workload**:



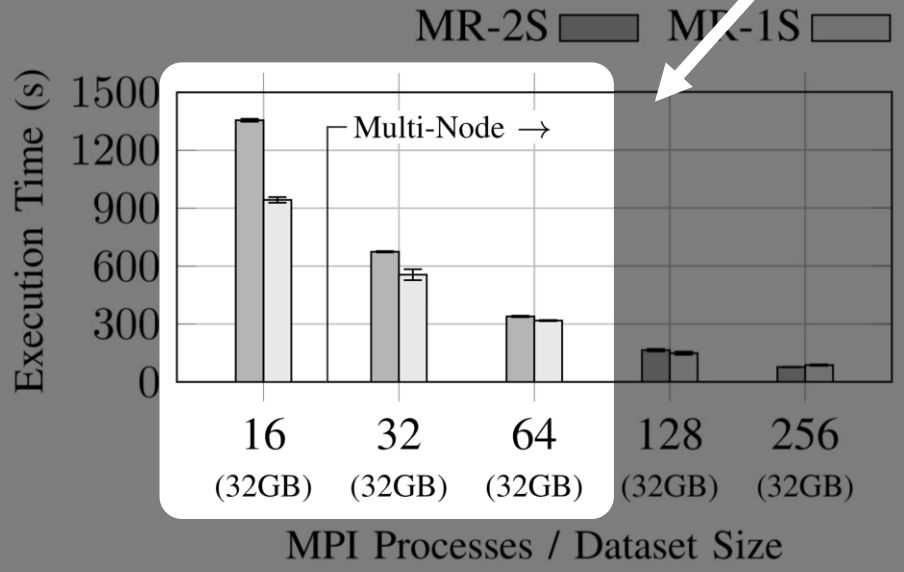
(c) **Strong Scaling Evaluation** under Unbalanced Workload

(d) **Weak Scaling Evaluation** under Unbalanced Workload

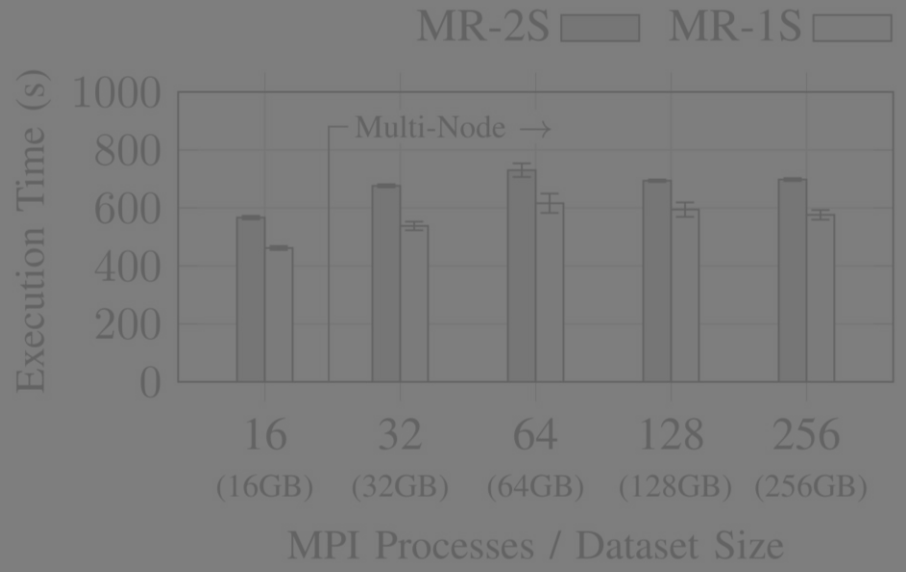
Experimental Results > Performance Evaluation

The following fig… MPI processes on Tegner for a fixed … **nced workload**:

If the workload is unbalanced, the average improvement is approximately 20.4% on average compared to MapReduce-2S



(c) **Strong Scaling Evaluation** under Unbalanced Workload

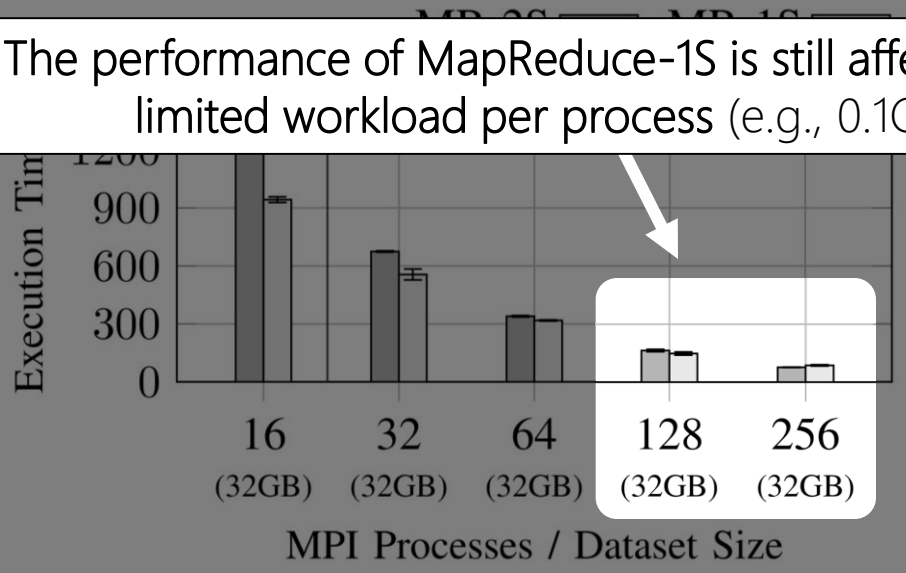


(d) **Weak Scaling Evaluation** under Unbalanced Workload

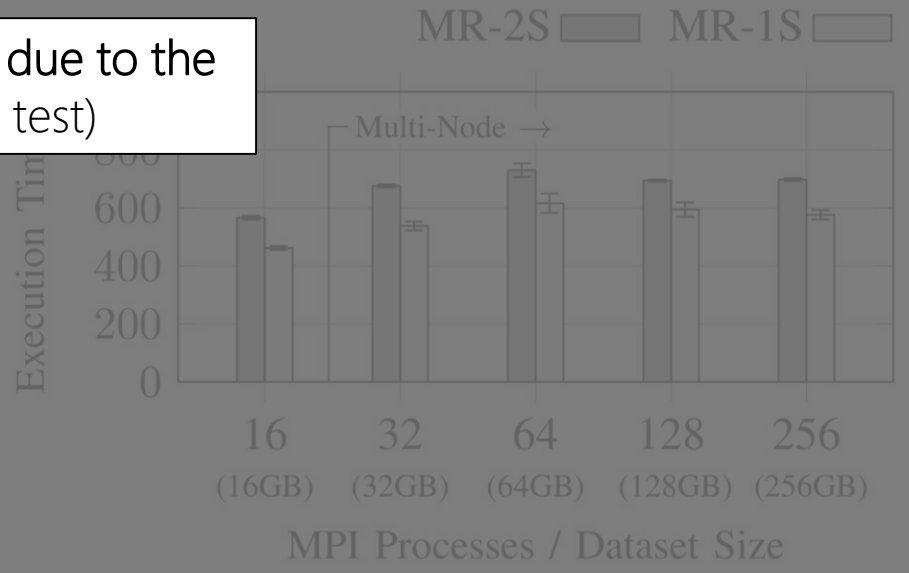
Experimental Results > Performance Evaluation

The following figure illustrates the performance of each approach by varying the number of MPI processes on Tegner for a fixed-size and variable-size input datasets (i.e., strong / weak scaling), **under unbalanced workload**:

The performance of MapReduce-1S is still affected, mainly due to the limited workload per process (e.g., 0.1GB on the last test)



(c) **Strong Scaling Evaluation** under Unbalanced Workload

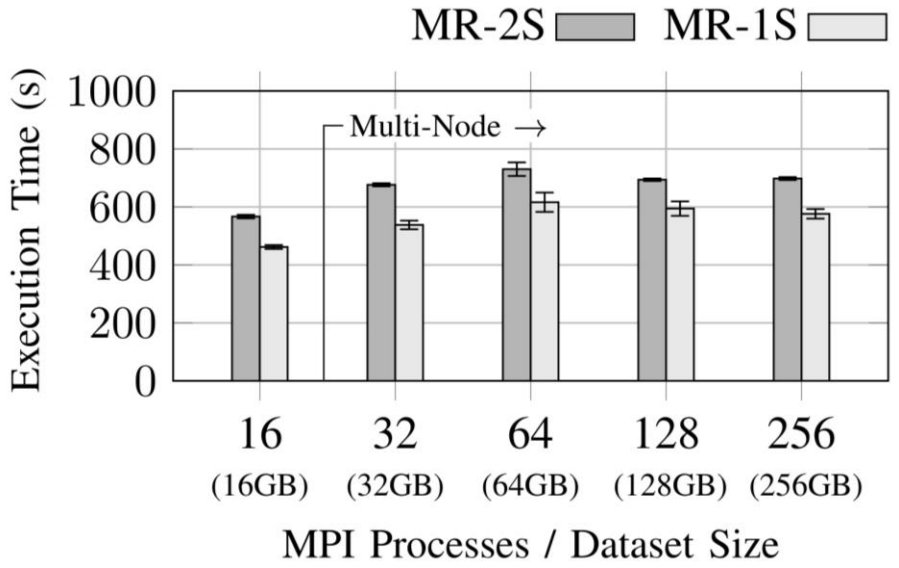


(d) **Weak Scaling Evaluation** under Unbalanced Workload

Experimental Results > Performance Evaluation

The following figure illustrates the performance of each approach by varying the number of MPI processes on Tegner for a fixed-size and variable-size input datasets (i.e., strong / weak scaling), **under unbalanced workload**:

(c) **Strong Scaling Evaluation** under Unbalanced Workload

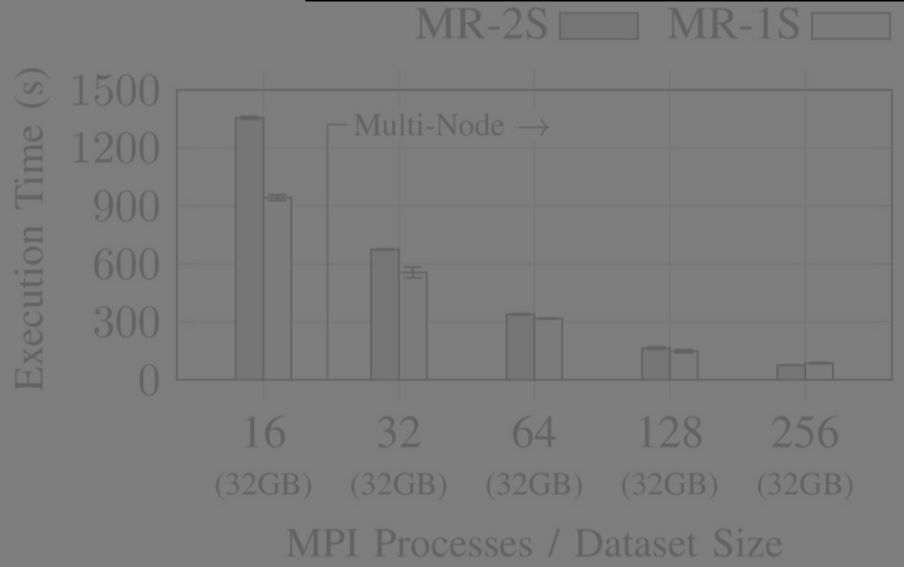


(d) **Weak Scaling Evaluation** under Unbalanced Workload

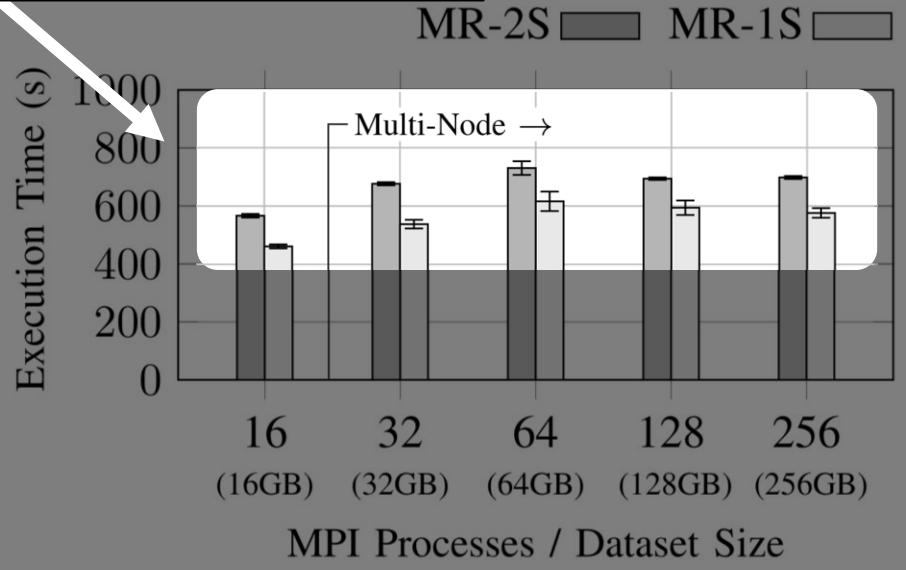
Experimental Results > Performance Evaluation

The following figure illustrates the performance of each approach by varying the number of MPI processes on Tegner for a fixed-s **unbalanced workload**:

The improvement on unbalanced workloads is 23.1% on average for MapReduce-1S (with a peak of 33.9%)



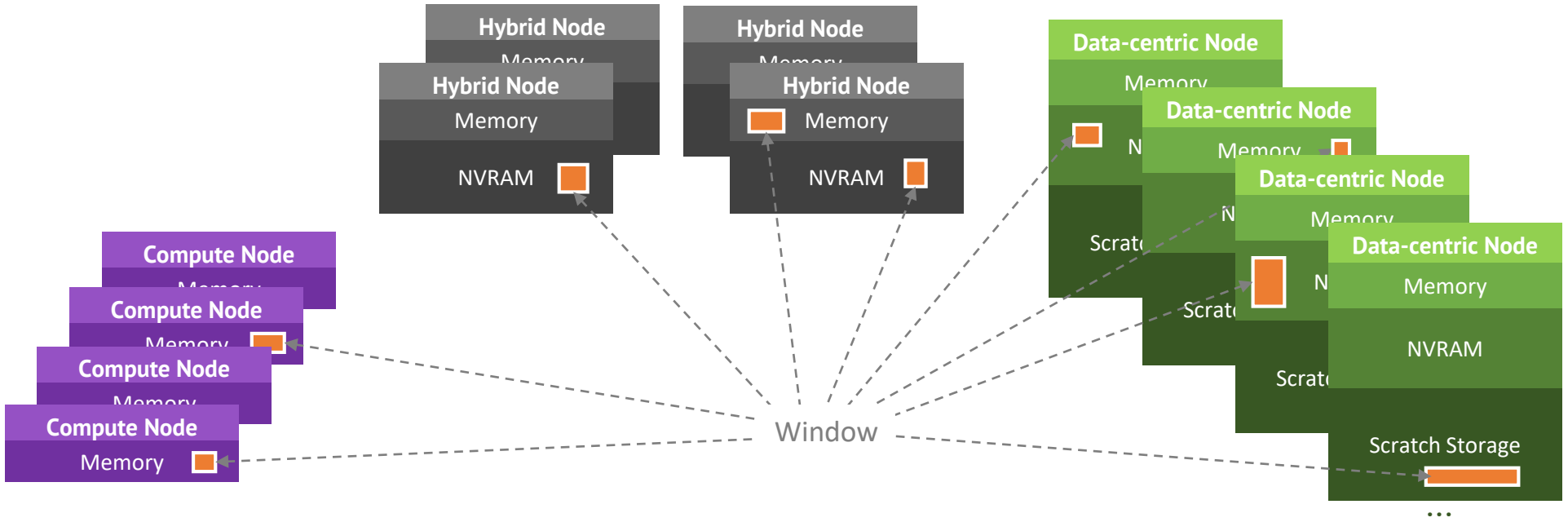
(c) **Strong Scaling Evaluation** under Unbalanced Workload



(d) **Weak Scaling Evaluation** under Unbalanced Workload

Background > MPI windows on storage

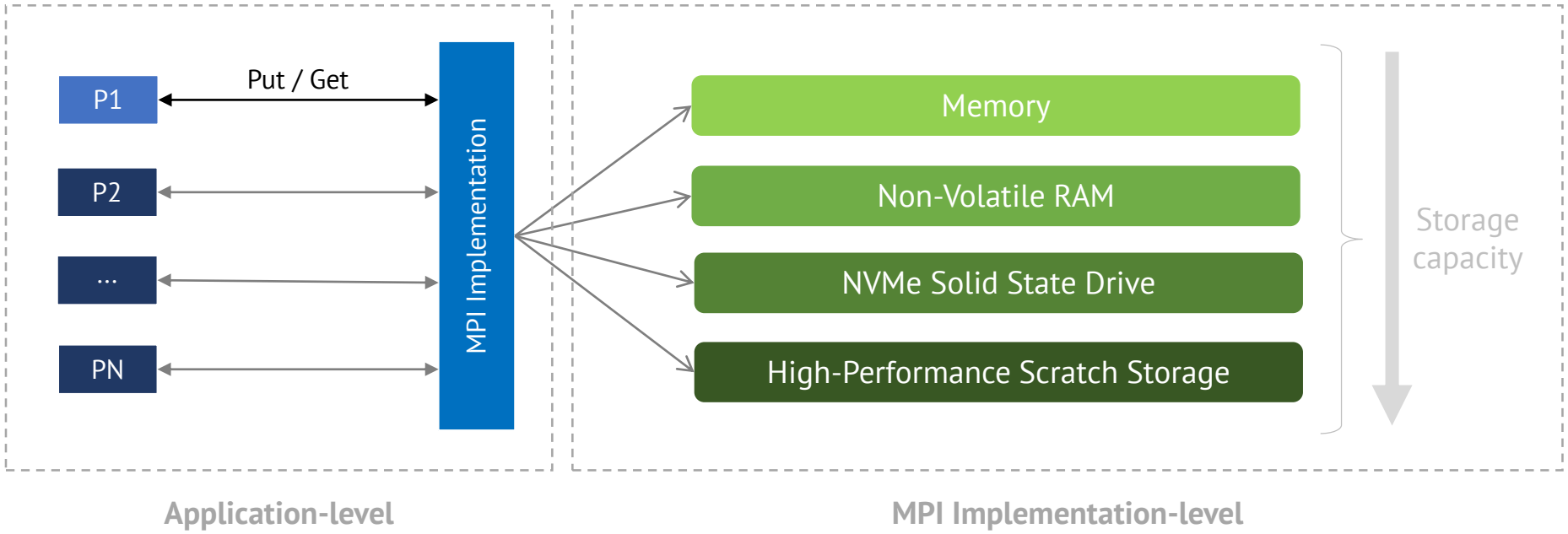
This on-going effort proposes to extend the concept of MPI window to storage. Hence, allowing programmers to map MPI windows to a wide-range of persistent storage devices:



S. Rivas-Gomez, R. Gioiosa, I. B. Peng, G. Kestor, S. Narasimhamurthy, E. Laure, and S. Markidis, "**MPI Windows on Storage for HPC Applications**," in Proceedings of the 24th European MPI Users' Group Meeting. ACM, 2017, p. 15. **[Best Paper Award]**

Background > Unified interface for conducting I/O

With MPI storage windows, **both memory and storage become accessible through the unified, familiar interface based of the MPI one-sided communication model:**



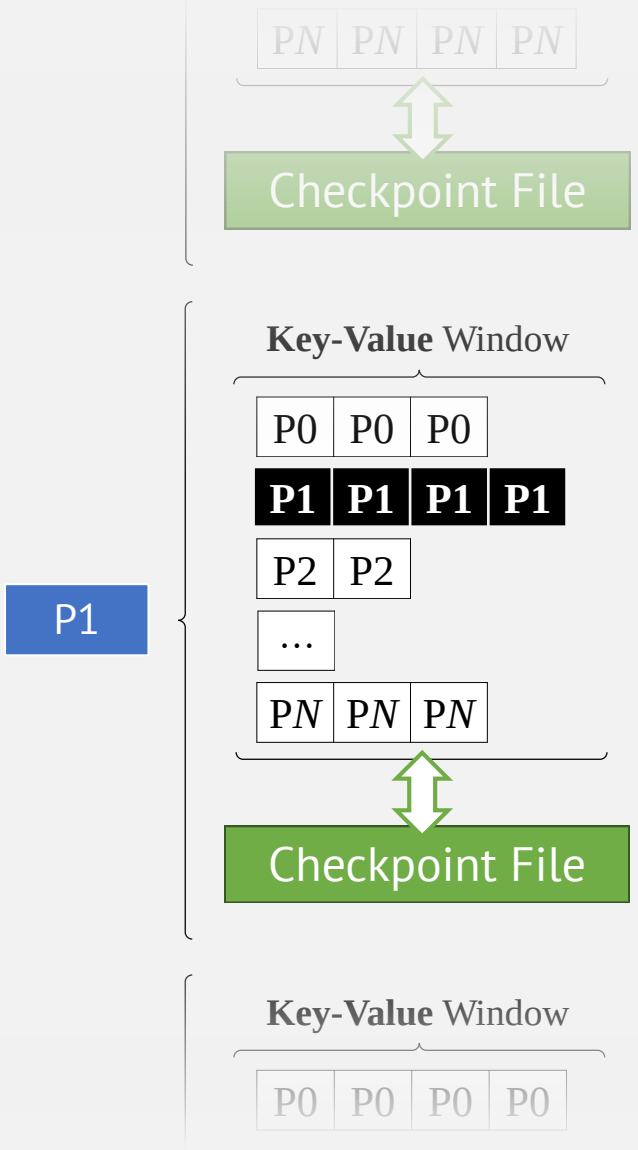
Evaluation > Transparent Checkpoint

With the arrival of the first wave of pre-Exascale machines, the chances for unexpected failures during the execution of parallel applications will considerably increase.

By integrating the concept of MPI storage windows, **we can define a novel checkpoint mechanism inside MapReduce-1S**.

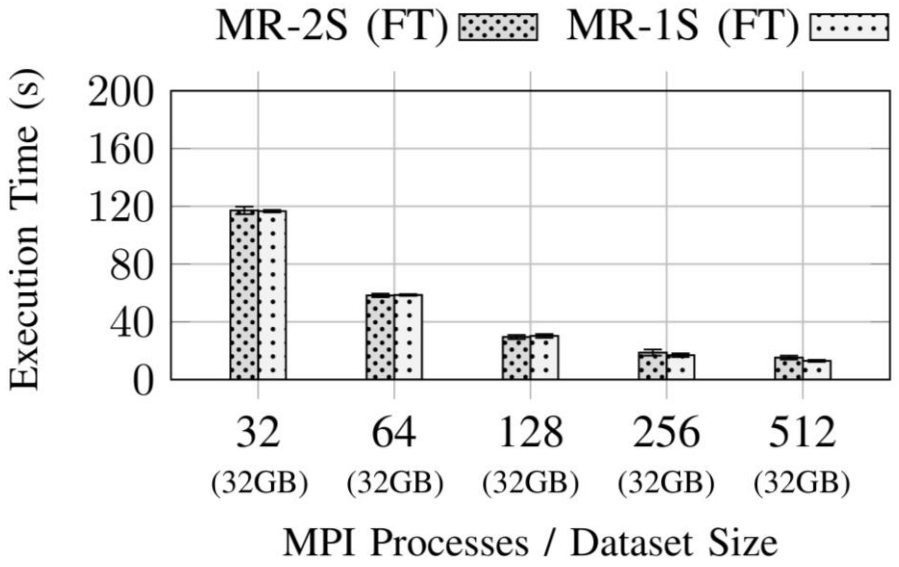
We introduce support for MPI storage windows and extend our approach to perform a window synchronization point after each Map task, as well as after the Reduce phase is completed.

In addition, **we extend MapReduce-2S to perform each checkpoint using MPI collective I/O instead.**



Experimental Results > Performance Evaluation

The following figure illustrates the checkpoint performance of each approach by varying the number of MPI processes on Tegner for a fixed-size input dataset (i.e., strong scaling), **under balanced workload**:

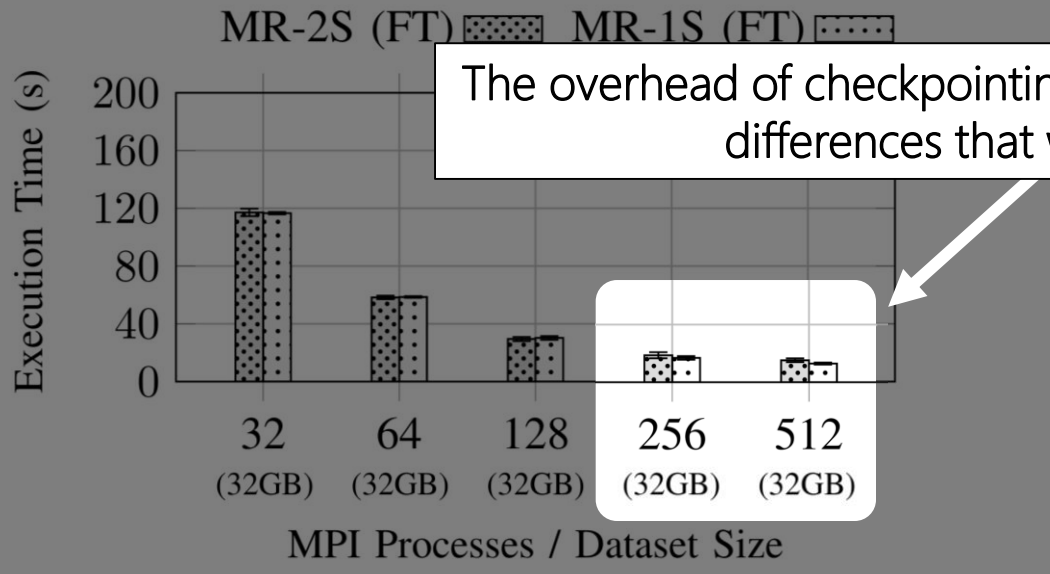


(a) **Strong Scaling Evaluation** with Checkpoint Support

(b) **Checkpoint Performance Overhead** (Strong Scaling)

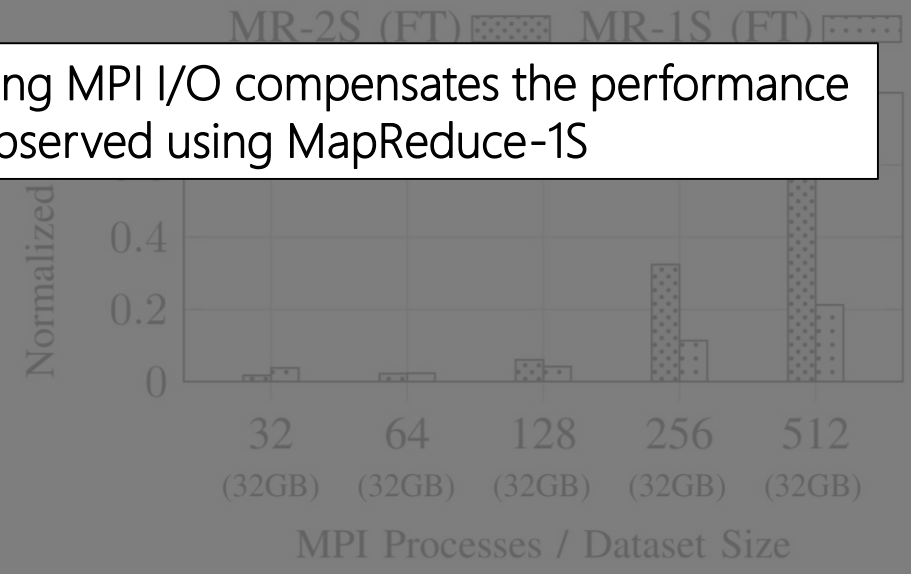
Experimental Results > Performance Evaluation

The following figure illustrates the checkpoint performance of each approach by varying the number of MPI processes on Tegner for a fixed-size input dataset (i.e., strong scaling), **under balanced workload**:



(a) **Strong Scaling Evaluation** with Checkpoint Support

The overhead of checkpointing using MPI I/O compensates the performance differences that we observed using MapReduce-1S

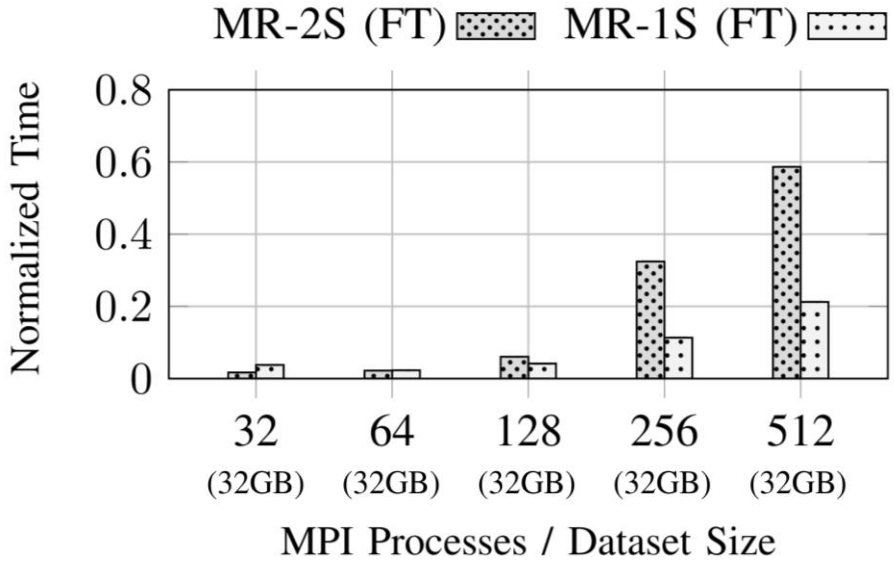


(b) **Checkpoint Performance Overhead** (Strong Scaling)

Experimental Results > Performance Evaluation

Using 512 processes, MapReduce-1S with checkpoint support is 17.6% faster in comparison with MapReduce-2S. In this particular case, **the checkpoint overhead is 21.2% for MapReduce-1S and 58.6% for MapReduce-2S**:

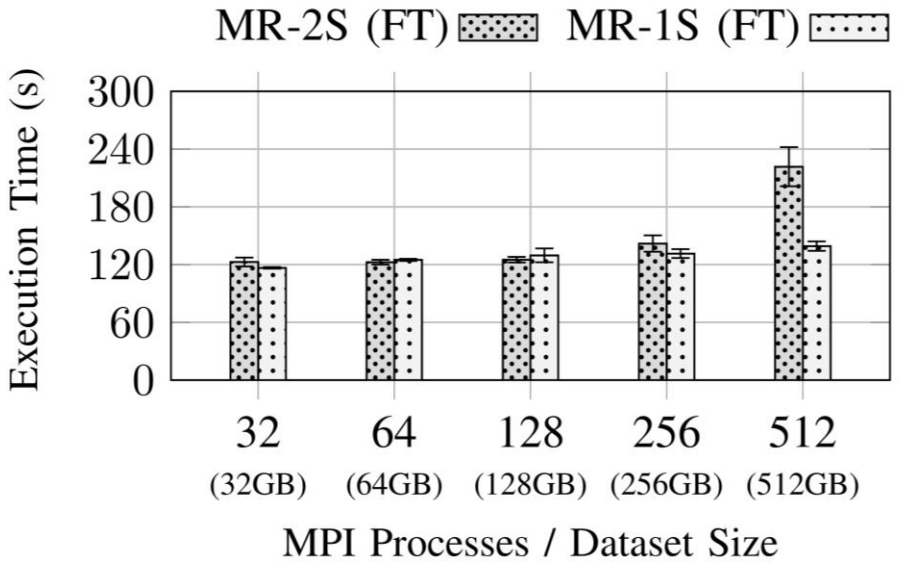
(a) **Strong Scaling Evaluation** with Checkpoint Support



(b) **Checkpoint Performance Overhead** (Strong Scaling)

Experimental Results > Performance Evaluation

The following figure illustrates the checkpoint performance of each approach by varying the number of MPI processes on Tegner for a variable-size input datasets (i.e., weak scaling), **under balanced workload**:



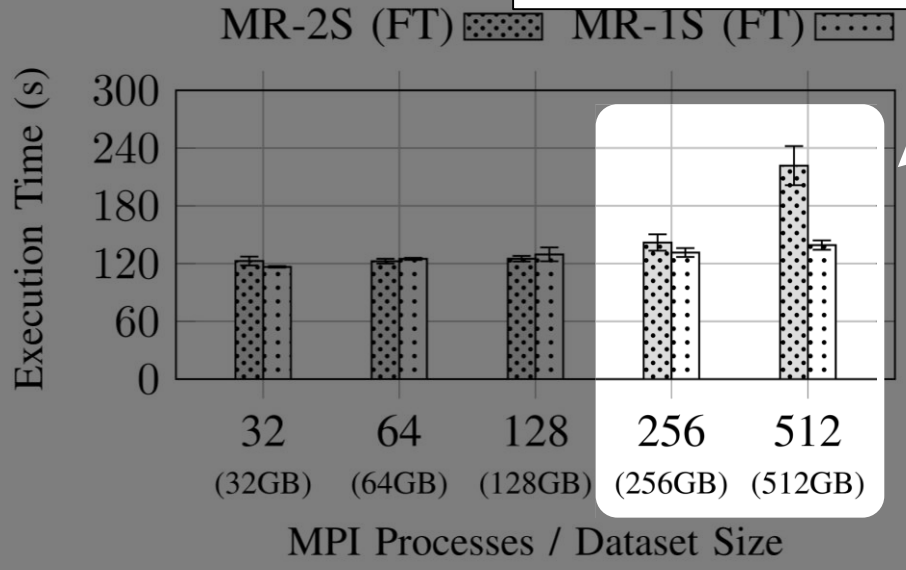
(c) **Weak Scaling Evaluation** with Checkpoint Support

(d) **Checkpoint Performance Overhead** (Weak Scaling)

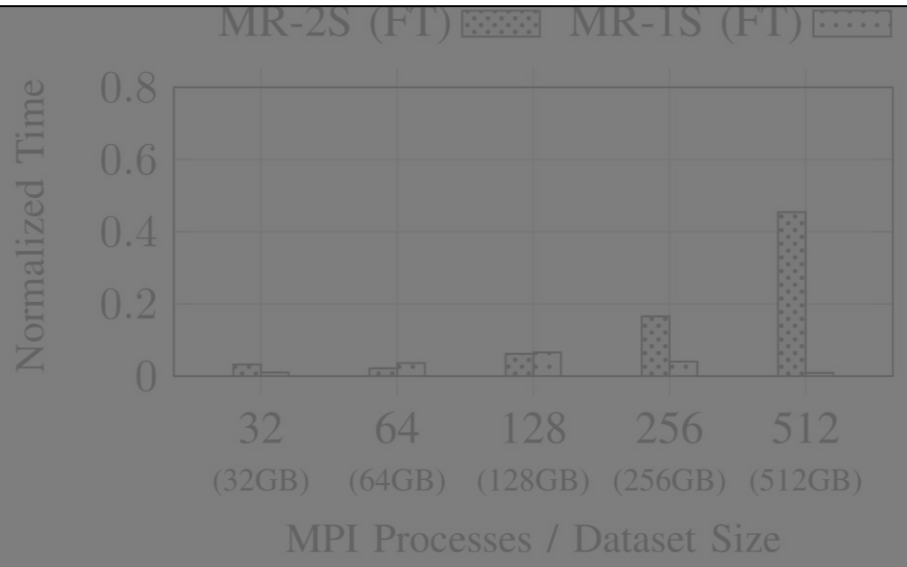
Experimental Results > Performance Evaluation

The following figure illustrates the checkpoint performance of each approach by varying the number of MPI processes on Tegner for a var

The overhead of checkpointing using MPI I/O is, once again, affected as the process count increases, making our approach more suitable



(c) **Weak Scaling Evaluation** with Checkpoint Support

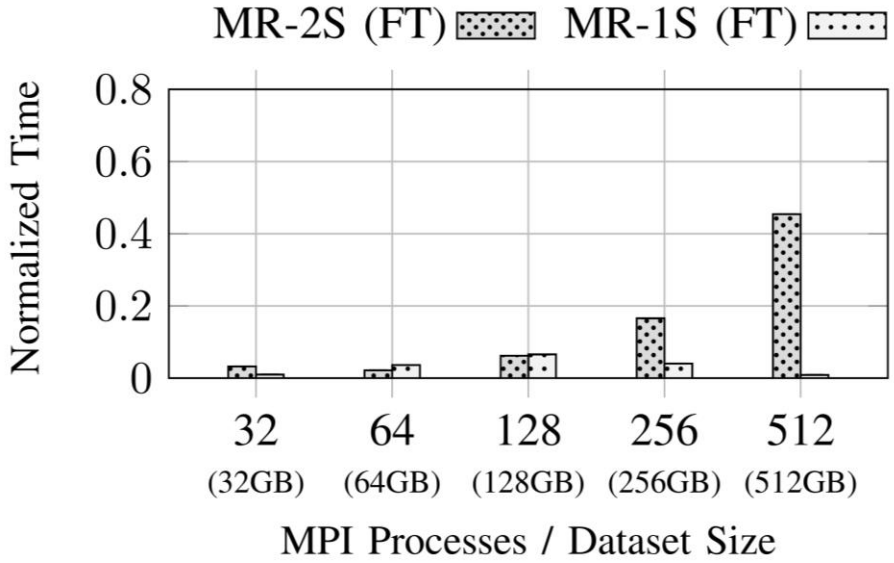


(d) **Checkpoint Performance Overhead** (Weak Scaling)

Experimental Results > Performance Evaluation

The performance of MapReduce-1S with checkpoint support is 59.3% faster in the last case of 512 processes, in comparison with MapReduce-2S. **The checkpoint overhead using MPI storage windows is only 3.8% on average**:

(c) **Weak Scaling Evaluation** with Checkpoint Support



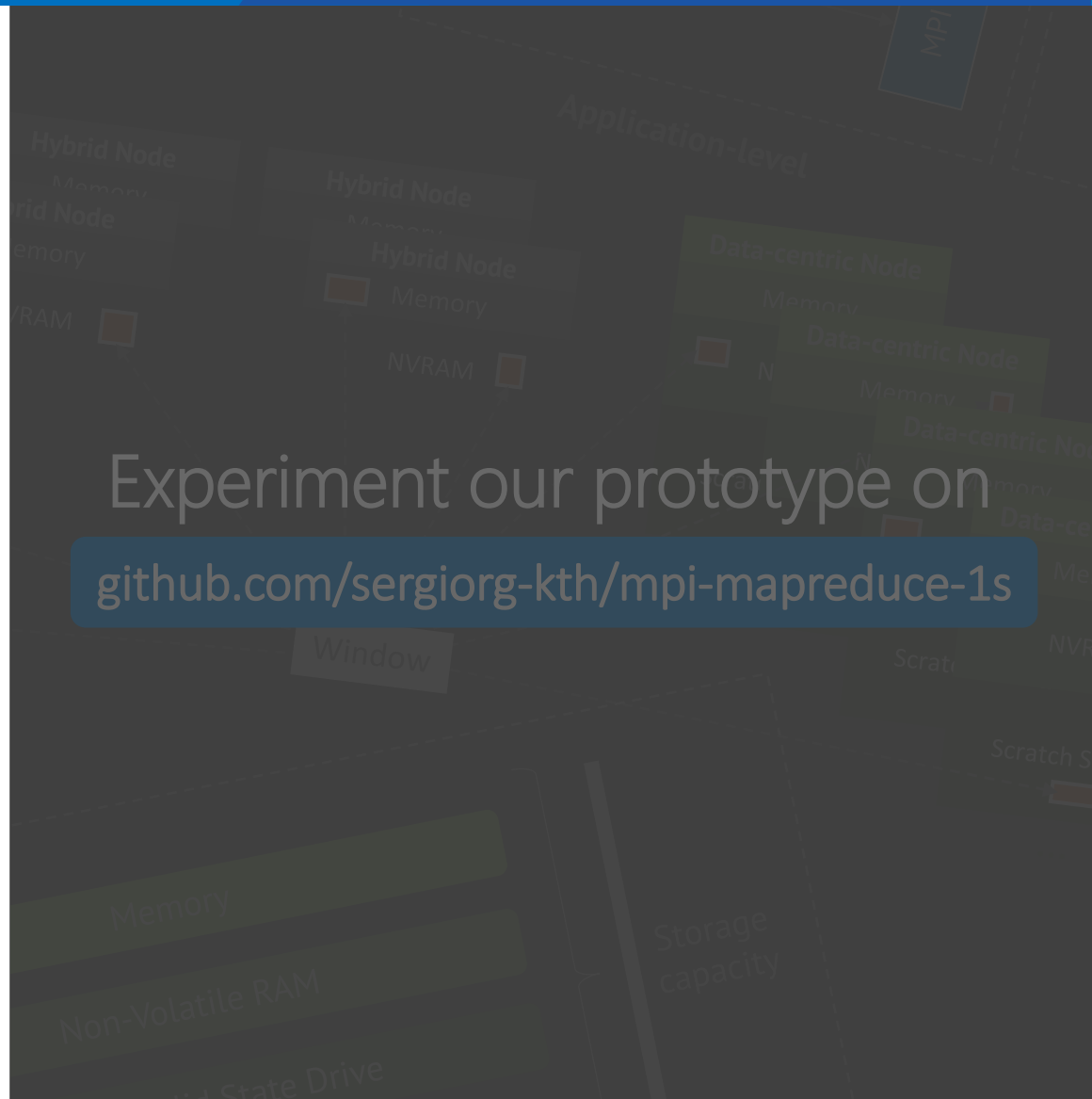
(d) **Checkpoint Performance Overhead** (Weak Scaling)

Conclusions

With the emergence of machine learning and data-centric applications on HPC, MapReduce has become one of the preferred programming models to hide the complexity of process and data parallelism.

We propose decoupled strategy for MapReduce:

- Hide the heterogeneity of upcoming HPC clusters.
- Overlap the Map and Reduce phases of MapReduce using MPI one-sided and non-blocking I/O.
- Benefits are limited on large-process counts and balanced workloads per process, but it does feature performance advantages on unbalanced workloads.
- Negligible performance degradation with checkpoint support using MPI storage windows.

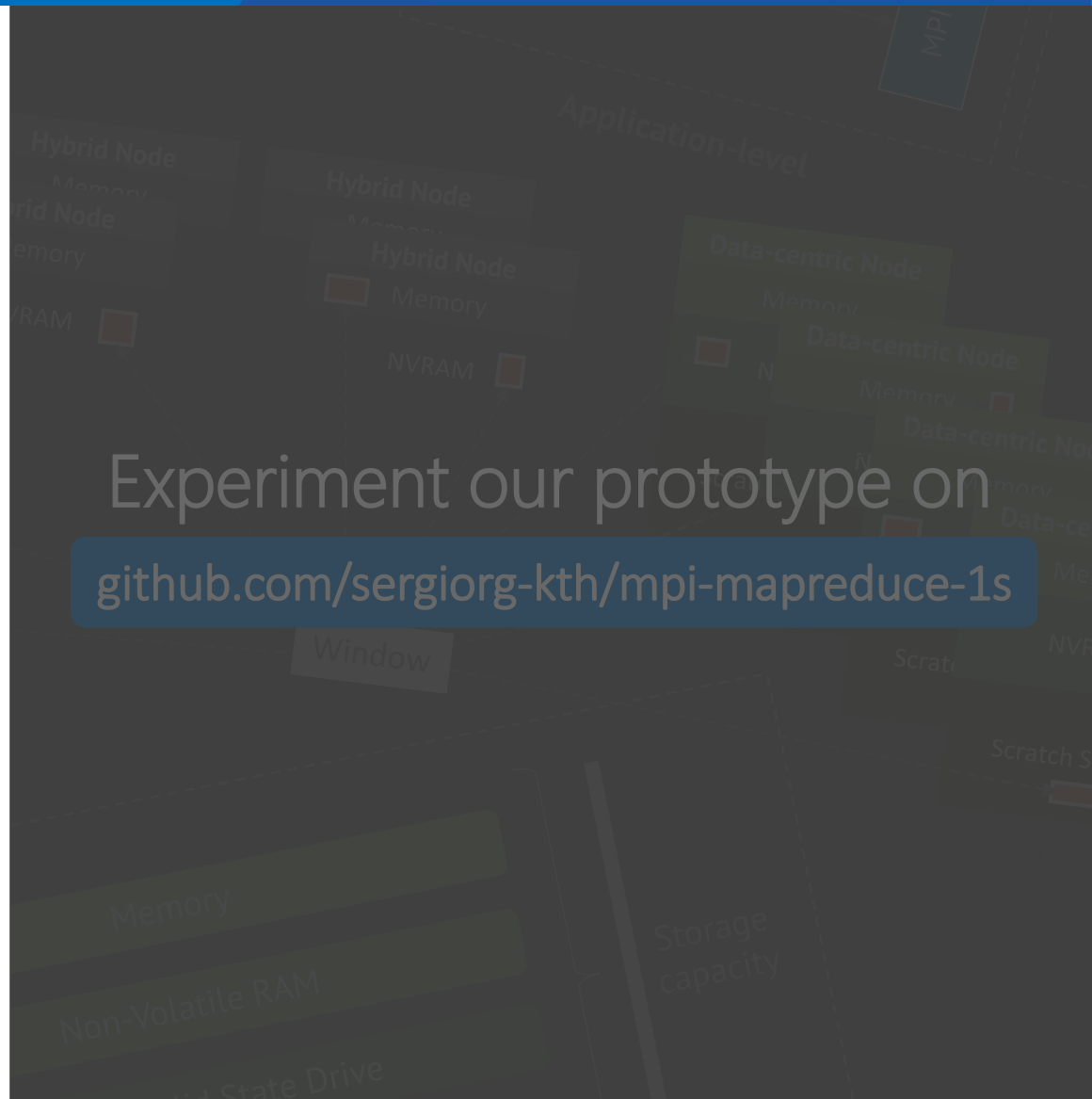


Conclusions

With the emergence of machine learning and data-centric applications on HPC, MapReduce has become one of the preferred programming models to hide the complexity of process and data parallelism.

We propose decoupled strategy for MapReduce:

- Hide the heterogeneity of upcoming HPC clusters.
- Overlap the Map and Reduce phases of MapReduce using MPI one-sided and non-blocking I/O.
- Benefits are limited on large-process counts and balanced workloads per process, but it does feature performance advantages on unbalanced workloads.
- Negligible performance degradation with checkpoint support using MPI storage windows.

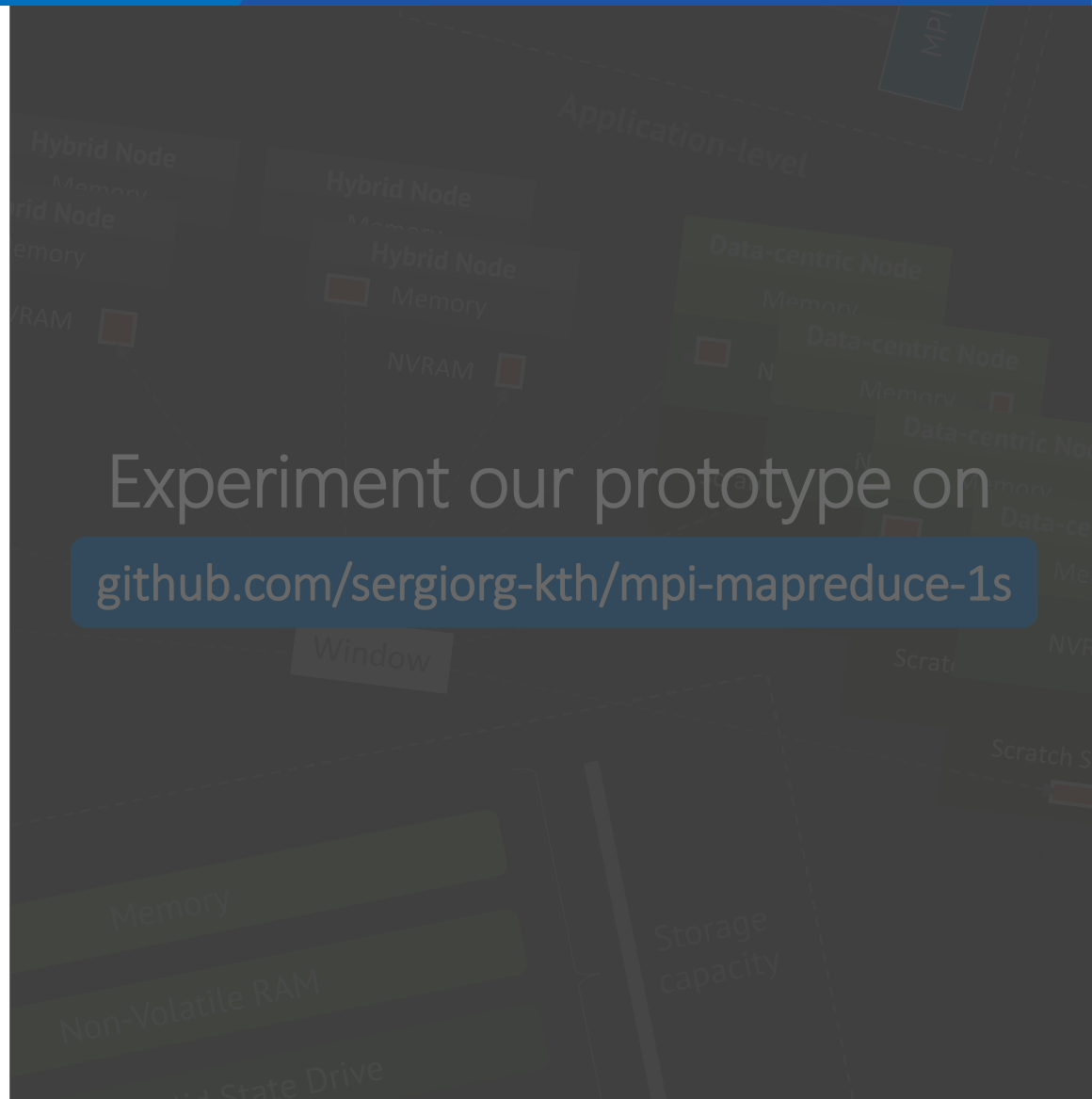


Conclusions

With the emergence of machine learning and data-centric applications on HPC, MapReduce has become one of the preferred programming models to hide the complexity of process and data parallelism.

We propose decoupled strategy for MapReduce:

- Hide the heterogeneity of upcoming HPC clusters.
- Overlap the Map and Reduce phases of MapReduce using MPI one-sided and non-blocking I/O.
- Benefits are limited on large-process counts and balanced workloads per process, but it does feature performance advantages on unbalanced workloads.
- Negligible performance degradation with checkpoint support using MPI storage windows.

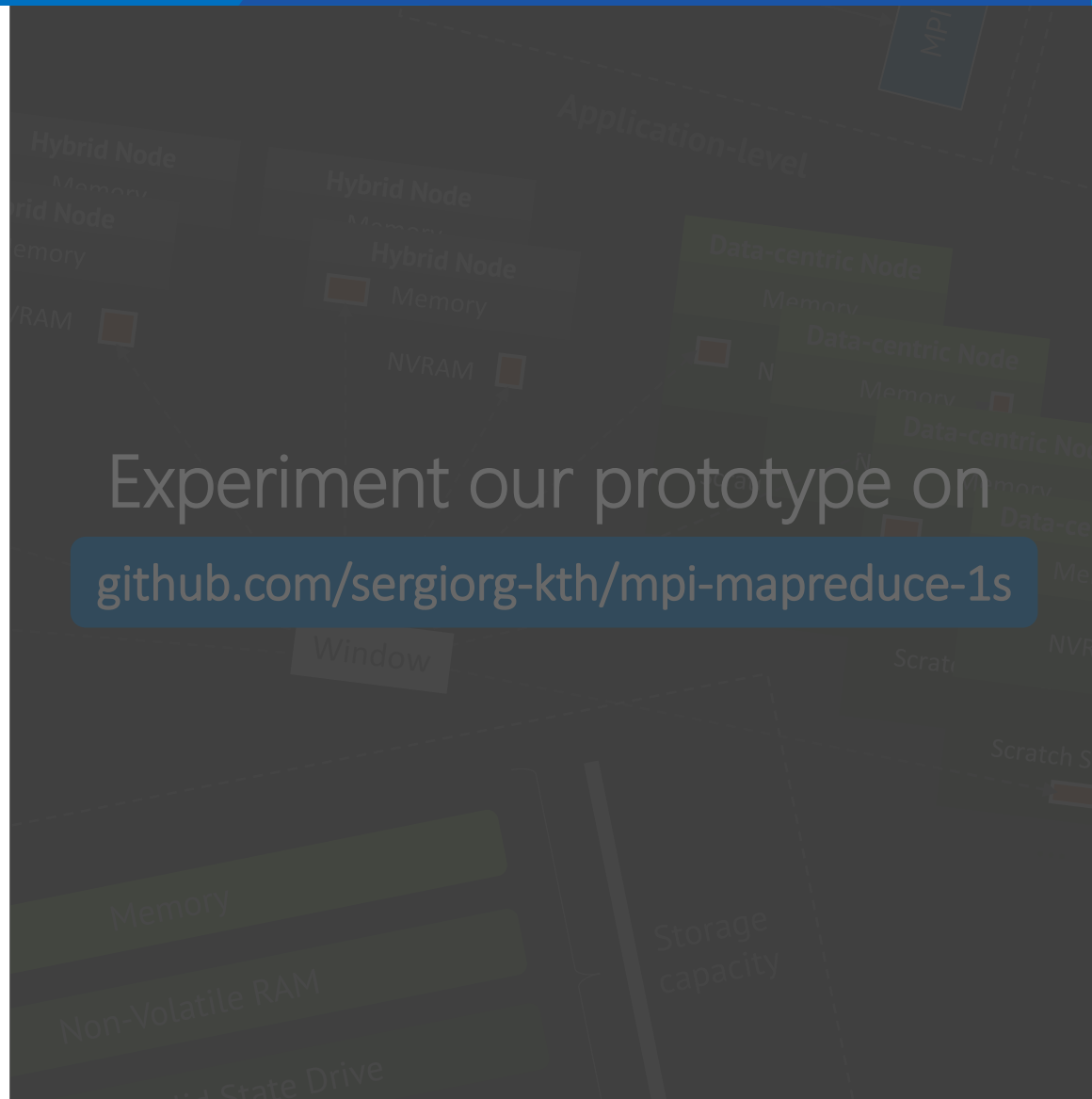


Conclusions

With the emergence of machine learning and data-centric applications on HPC, MapReduce has become one of the preferred programming models to hide the complexity of process and data parallelism.

We propose decoupled strategy for MapReduce:

- Hide the heterogeneity of upcoming HPC clusters.
- Overlap the Map and Reduce phases of MapReduce using MPI one-sided and non-blocking I/O.
- Benefits are limited on large-process counts and balanced workloads per process, but it does feature performance advantages on unbalanced workloads.
- Negligible performance degradation with checkpoint support using MPI storage windows.

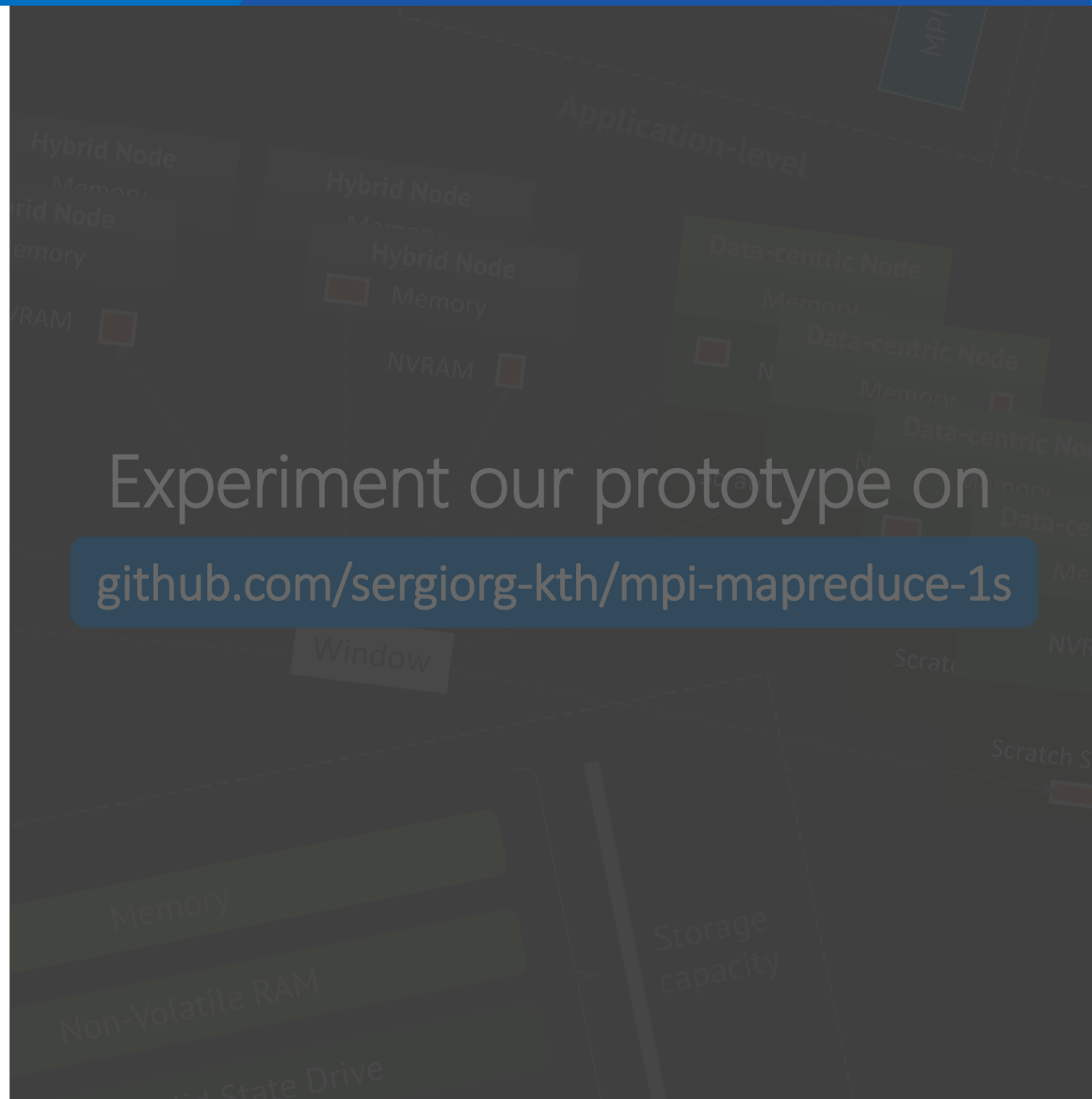


Conclusions

With the emergence of machine learning and data-centric applications on HPC, MapReduce has become one of the preferred programming models to hide the complexity of process and data parallelism.

We propose decoupled strategy for MapReduce:

- Hide the heterogeneity of upcoming HPC clusters.
- Overlap the Map and Reduce phases of MapReduce using MPI one-sided and non-blocking I/O.
- Benefits are limited on large-process counts and balanced workloads per process, but it does feature performance advantages on unbalanced workloads.
- Negligible performance degradation with checkpoint support using MPI storage windows.

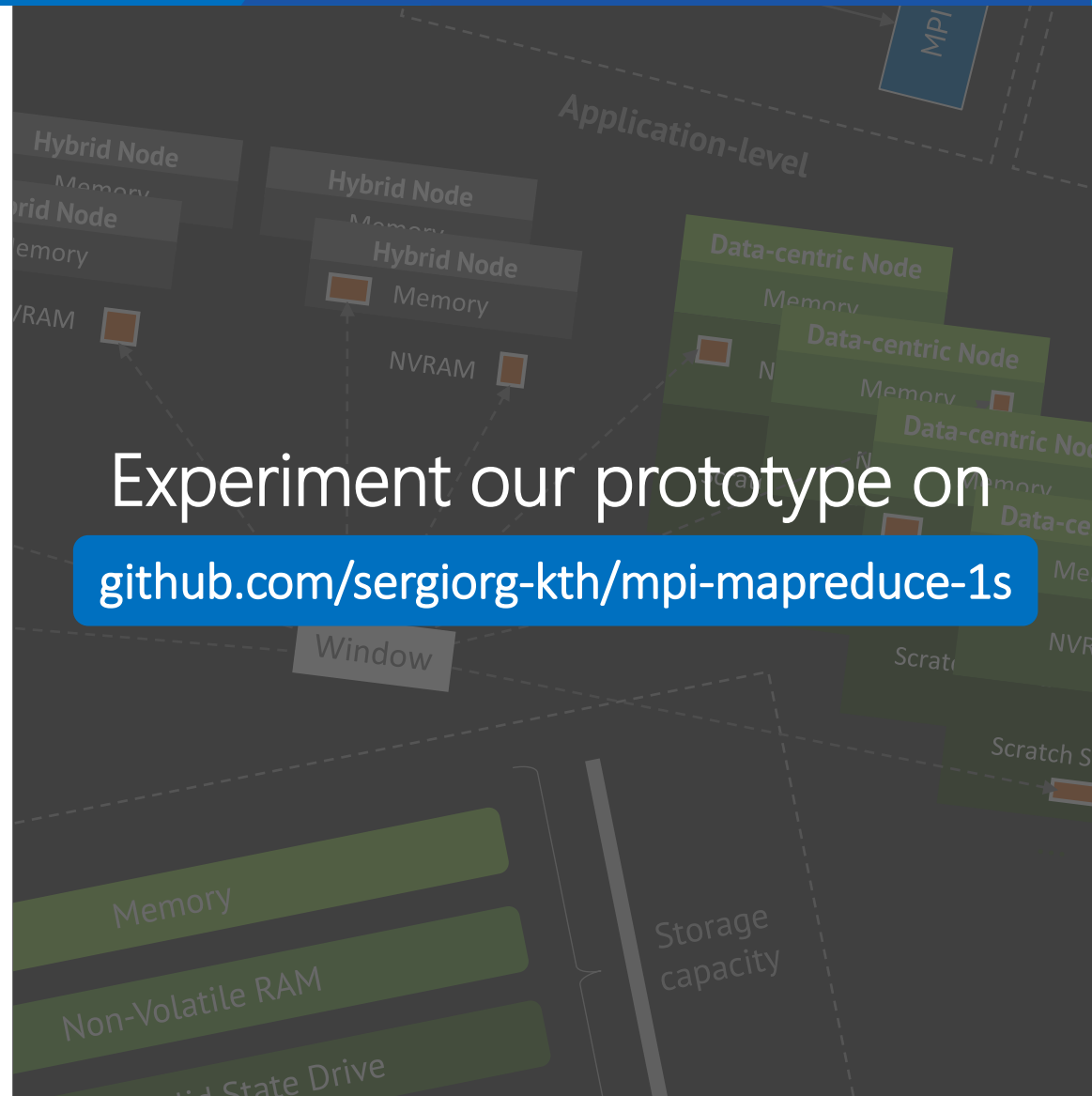


Conclusions

With the emergence of machine learning and data-centric applications on HPC, MapReduce has become one of the preferred programming models to hide the complexity of process and data parallelism.

We propose decoupled strategy for MapReduce:

- Hide the heterogeneity of upcoming HPC clusters.
- Overlap the Map and Reduce phases of MapReduce using MPI one-sided and non-blocking I/O.
- Benefits are limited on large-process counts and balanced workloads per process, but it does feature performance advantages on unbalanced workloads.
- Negligible performance degradation with checkpoint support using MPI storage windows.



European Commission

Horizon 2020
European Union funding
for Research & Innovation

Acknowledgements

This work is funded by the European Commission through the SAGE Project
[Grant agreement no. 671500 | More information on http://www.sagestorage.eu]

Thank you for coming!

Questions?