
BasicAnalysis Documentation

Release 2026.09.17

BSC Performance Tools

Sep 17, 2026

CONTENTS:

1	Introduction	3
1.1	Parallel execution environments	3
1.2	Analysis workflow	4
2	Getting Started	5
2.1	Requirements	5
2.2	Python dependencies	5
2.3	Installation	6
2.4	Running a first analysis	6
2.5	Analysis results	6
2.6	Next steps	7
3	Running BasicAnalysis	9
3.1	Standard workflow	9
3.2	Metric workflow	9
3.3	Scaling model	10
3.4	Selecting the MPI+GPU metric model	10
3.5	Dimemas simulation	11
3.6	Trace-mode detection	11
3.7	Trace ordering	11
3.8	Parallel trace analysis	12
3.9	Maximum trace size	12
3.10	Debug information	12
3.11	Version information	12
3.12	Command-line help	12
4	Staged Analysis Workflow	13
4.1	Trace analysis	14
4.2	Merging raw data	15
4.3	Computing metrics from merged data	16
4.4	Reanalyzing failed traces	16
4.5	Complete example	17
4.6	Choosing between the workflows	17
5	Performance Analysis Methodology	19
5.1	Hierarchical efficiency analysis	19
5.2	Parallel Runtime Model	22
5.3	Runtime-Specific Analysis	25
5.4	Execution Domains	29
5.5	I/O analysis	32
5.6	Scaling analysis	33
5.7	Complementary analytical views	34
6	Performance Metrics	37
6.1	From traces to efficiency metrics	37

6.2	Simple Metrics	39
6.3	Hybrid MPI+X Metrics for CPU-Based Applications	43
6.4	Hybrid MPI+GPU Metrics	49
6.5	OpenMP Runtime-Specific Metrics	60
6.6	I/O Metrics	63
6.7	General Metrics	66
6.8	Metric Availability	67
7	Performance Report	69
7.1	Report Interface	69
7.2	Execution Overview	70
7.3	Parallel Runtime Model	71
7.4	Runtime-Specific Analysis	72
7.5	Execution Domains	72
7.6	I/O Analysis	74
7.7	Scaling	75
7.8	Metric Details	76
7.9	Comparing Complementary Views	77
7.10	Exporting Report Content	77
7.11	Following the Analysis Workflow	79
8	Output	81
8.1	Performance Report	81
8.2	Numerical Output	81
8.3	Static Visualizations	82
8.4	Intermediate Analysis Files	82
9	Limitations	85
9.1	Simulation-Derived Metrics	85
9.2	Number of Parallel Runtimes	85
9.3	Hierarchical Execution Model	85
9.4	GPU Execution-Domain Analysis	86
9.5	I/O Analysis	86

BasicAnalysis automates the extraction of POP performance metrics from Paraver traces. These metrics help diagnose how efficiently a parallel application is running with respect to performance factors such as load balance, communication, and computation scalability, and help identify factors that may limit application scalability.

The tools extracts the performance data required to compute the efficiency metrics and organizes the results into complementary analytical views. The generated reports expose the relationships between the metrics and provide interpretation guidance, possible causes of efficiency losses, and possible next steps for further performance analysis.

INTRODUCTION

BasicAnalysis automates the extraction of POP performance metrics from Paraver traces. POP metrics help diagnose how efficiently a parallel application is running with respect to basic performance factors such as load balance and communication, as well as identify which factors may limit application scalability.

The tool automatically extracts from the traces the performance data required to compute the efficiency metrics and organizes the results into complementary views. These views expose the relationships between the metrics and provide different perspectives on the performance factors represented by the model.

The main analytical views provided by BasicAnalysis are:

- **Execution Overview**, which summarizes the analyzed traces, execution configurations, parallel resources, and general performance characteristics.
- **Parallel Runtime Model**, which presents the main application-level efficiency factors and their hierarchical relationships and, for hybrid applications, shows the contribution of the active parallel runtimes to Parallel Efficiency.
- **Runtime-Specific Analysis**, which provides additional metrics for the active parallel runtimes when runtime-specific information is available.
- **Execution Domains**, which provides complementary Host and Device views for accelerator applications and helps identify where accelerator-related inefficiencies manifest.
- **Scaling**, which shows how efficiency metrics and other performance indicators evolve across the analyzed execution configurations.

The generated interactive and printable reports preserve the relationships between the efficiency metrics and provide metric definitions, interpretation guidance, possible causes of efficiency losses, and possible next steps for further performance analysis.

1.1 Parallel execution environments

The efficiency metrics implemented by BasicAnalysis are based on general performance factors of parallel executions and are therefore not restricted to a specific parallel programming model. Since most of the metrics are derived from useful computation and execution time, BasicAnalysis can be applied to different parallel execution environments represented in Paraver traces.

Examples of parallel programming models and combinations currently handled by BasicAnalysis include:

- MPI
- OpenMP
- MPI+OpenMP
- MPI+CUDA
- MPI+HIP

Depending on the parallel execution environment, BasicAnalysis may provide additional runtime-specific metrics and decompositions. The exact set of metrics available also depends on the information contained in the trace and the selected analysis model.

1.2 Analysis workflow

BasicAnalysis can be executed directly for a set of Paraver traces or through a staged workflow in which trace analysis, raw-data merging, and metric computation are performed independently.

The standard execution workflow is introduced in *Getting Started*, while the staged workflow is described in detail in *Staged Analysis Workflow*.

GETTING STARTED

This chapter describes the basic requirements for running **BasicAnalysis** and shows how to perform a first analysis from a set of Paraver traces.

For a complete description of the command-line options and execution modes, see *Running BasicAnalysis*.

2.1 Requirements

BasicAnalysis requires Python 3 and relies on tools from the BSC Performance Tools ecosystem to extract performance information and, when required, simulate idealized executions.

The main external requirements are:

- **Paraver** / **paramedir**, used to extract performance data from Paraver traces.
- **Dimemas**, used when simulation is required to compute communication efficiency submetrics such as Serialization Efficiency and Transfer Efficiency.

The tools are available from:

- Paraver: <https://tools.bsc.es/paraver>
- Dimemas: <https://tools.bsc.es/dimemas>

The corresponding executables must be available through the PATH environment variable. A typical configuration is:

```
export PATH=<paraver-install-dir>/bin:$PATH
export PARAVER_HOME=<paraver-install-dir>

export PATH=<dimemas-install-dir>/bin:$PATH
export DIMEMAS_HOME=<dimemas-install-dir>
```

2.2 Python dependencies

BasicAnalysis requires Python 3.

Additional Python packages are used for data processing, plotting, and report generation. The required packages and their installation are described with the BasicAnalysis distribution.

Some output functionality may depend on optional external tools. In particular, generation of the printable PDF performance report requires a supported Chromium-compatible browser.

2.2.1 PDF report export

BasicAnalysis generates an interactive HTML performance report as part of the analysis results.

A printable PDF version of the report can be generated on demand using the **Export** function available in the interactive report. PDF generation is not performed automatically during a BasicAnalysis execution.

The PDF export requires a supported Chromium-compatible browser to be available on the system, such as:

- chromium
- chromium-browser
- google-chrome
- google-chrome-stable

This separation allows BasicAnalysis to run on HPC systems where a compatible browser may not be installed. The interactive HTML report and the remaining analysis outputs are generated independently of PDF export.

2.3 Installation

BasicAnalysis does not require a separate installation procedure.

Clone or copy the BasicAnalysis repository to the desired location. The directory containing the BasicAnalysis executable scripts can optionally be added to PATH:

```
export PATH=<basicanalysis-dir>:$PATH
```

2.4 Running a first analysis

The standard BasicAnalysis workflow analyzes one or more Paraver traces. The tool accepts both uncompressed .prv traces and compressed .prv.gz traces and computes the corresponding efficiency metrics and other performance indicators, tables, plots, and reports.

BasicAnalysis can be executed with:

```
modelfactors.py [options] <list-of-traces>
```

The trace list can contain explicit trace filenames, multiple traces, or wildcard expressions. Only valid Paraver traces are retained for analysis.

For example:

```
modelfactors.py application.prv
```

Compressed trace:

```
modelfactors.py application.prv.gz
```

Several execution configurations can be analyzed together:

```
modelfactors.py trace_1.prv trace_2.prv trace_3.prv
```

Wildcards can also be used:

```
modelfactors.py *.prv
```

When several traces are analyzed, BasicAnalysis can compare their performance and evaluate how the efficiency factors evolve across the execution configurations.

2.5 Analysis results

A standard BasicAnalysis execution can produce:

- raw performance data;
- efficiency and model-factor tables;
- CSV files;

- performance and scalability plots; and
- an interactive HTML performance report.

The performance report organizes the analysis into complementary views such as Execution Overview, Parallel Runtime Model, Runtime-Specific Analysis, Execution Domains when applicable, and Scaling when several configurations are available.

The interactive report can also be exported as a printable PDF when the required browser support is available.

A detailed description of the generated files is provided in [Output](#).

2.6 Next steps

After completing a first analysis:

- See [Running BasicAnalysis](#) for the available execution options and analysis configuration.
- See [Staged Analysis Workflow](#) when traces should be analyzed independently and merged before computing the final metrics.
- See [Performance Analysis Methodology](#) to understand the BasicAnalysis performance-analysis methodology.
- See [Performance Report](#) for guidance on reading the generated performance report.
- See [Performance Metrics](#) for the definitions of the efficiency metrics.

RUNNING BASICANALYSIS

This chapter describes how to run the standard BasicAnalysis workflow and configure the analysis.

For workflows in which trace analysis and metric computation need to be performed independently, see [Staged Analysis Workflow](#).

3.1 Standard workflow

The standard BasicAnalysis workflow is executed with `modelfactors.py`:

```
modelfactors.py [options] <list-of-traces>
```

The command accepts one or more Paraver traces in either uncompressed `.prv` format or compressed `.prv.gz` format.

Trace filenames can be specified explicitly:

```
modelfactors.py trace_1.prv trace_2.prv trace_3.prv
```

Wildcard expressions can also be used, for example:

```
modelfactors.py *.prv
```

or for compressed traces:

```
modelfactors.py *.prv.gz
```

BasicAnalysis automatically filters the input list to retain valid Paraver traces, including compressed traces, and excludes traces generated internally by simulation.

By default, traces are ordered according to their parallel configuration before the metrics are computed. When several execution configurations are provided, BasicAnalysis uses them to evaluate performance and scalability trends.

3.2 Metric workflow

BasicAnalysis automatically selects the metric workflow according to the programming models detected in the analyzed traces.

If all traces correspond to simple execution models, BasicAnalysis uses the simple metric workflow. If at least one trace contains hybrid parallelism, BasicAnalysis uses the hybrid metric workflow.

Therefore, users normally do not need to select the metric workflow explicitly.

The metric workflow can be controlled manually with:

```
-m, --metrics {simple,hybrid}
```

The available modes are:

simple

Force the use of the simple metric workflow.

hybrid

Allow BasicAnalysis to use the hybrid metric workflow when required by the detected trace modes. If all analyzed traces correspond to simple execution models, BasicAnalysis automatically uses the simple metric workflow.

The default is **hybrid**.

3.3 Scaling model

The scaling model can be selected with:

```
-s, --scaling {weak,strong,auto}
```

The available modes are:

strong

Use the strong-scaling formulation when computing scalability metrics.

weak

Use the weak-scaling formulation when computing scalability metrics.

auto

Let BasicAnalysis determine the scaling behavior from the analyzed executions.

The default is **auto**.

The scaling model affects the computation and interpretation of scalability metrics. The detected and selected scaling models are reported in the **Scaling** section of the performance report.

See *Performance Analysis Methodology* and *Performance Metrics* for the definition and interpretation of the scalability metrics.

3.4 Selecting the MPI+GPU metric model

For MPI+GPU applications, BasicAnalysis computes both the classic multiplicative model and the TALP-based model.

The model displayed in the standard output can be selected with:

```
-pop-model, --pop_model_to_apply {classic,talp}
```

The available models are:

classic

Display the classic multiplicative model in the standard output.

talp

Display the TALP-based model in the standard output.

The default is **talp**.

This option affects the model presented in the standard output. The interactive Performance Report provides access to both models, allowing users to inspect the complementary information provided by each one.

The classic and TALP-based models, including the Host and Device execution-domain analysis associated with the TALP-based model, are described in *Performance Analysis Methodology* and *Performance Metrics*.

3.5 Dimemas simulation

Some communication-efficiency metrics require information obtained from an idealized execution simulated with Dimemas.

Simulation can be disabled with:

```
-skip-simul, --skip-simulation
```

When this option is used, BasicAnalysis does not run the Dimemas simulation. Metrics that depend on simulated execution information may consequently be reported as unavailable.

Additional simulation options are available for traces containing OpenMP or CUDA activity:

```
-somp, --simulation_openmp
-scuda, --simulation_cuda
```

--simulation_openmp

Enable simulation of OpenMP events.

--simulation_cuda

Enable simulation of CUDA events.

The role of Dimemas, the idealized execution, and the metrics that depend on simulation are described in *Performance Metrics*.

3.6 Trace-mode detection

BasicAnalysis automatically determines the programming and tracing mode of the analyzed traces.

The source used for trace-mode detection can be selected with:

```
-tmd, --trace_mode_detection {pcf,prv}
```

pcf

Detect the trace mode using the Paraver configuration file. This is the default.

prv

Inspect the trace itself to determine the available instrumentation. This mode is useful for customized traces, such as filtered or cut traces, whose contents may no longer correspond exactly to the original .pcf description.

The default is pcf.

3.7 Trace ordering

By default, BasicAnalysis orders the analyzed traces according to their parallel configuration.

This behavior is controlled with:

```
-ord, --order_traces {yes,not}
```

yes

Order the traces before performing the comparative analysis.

not

Preserve the input trace order.

The default is yes.

Trace ordering is particularly relevant when several configurations are analyzed because the first execution is used as the reference for several relative performance and scalability metrics.

3.8 Parallel trace analysis

Independent traces can be analyzed concurrently to reduce the total analysis time.

The number of concurrent workers is controlled with:

```
--jobs <number|auto>
```

1

Analyze traces sequentially. This is the default.

<number>

Analyze up to the specified number of traces concurrently.

auto

Let BasicAnalysis determine an appropriate number of concurrent workers according to the available resources.

The estimated memory available to each worker can be controlled with:

```
--mem-per-worker-gb <value>
```

This option overrides the automatic memory estimate used when determining parallel trace-processing capacity.

Parallel processing applies to independent trace analyses; it does not change the performance metrics computed for each trace.

3.9 Maximum trace size

The maximum input trace size accepted by BasicAnalysis can be controlled with:

```
-ms, --max_trace_size <MiB>
```

The default maximum size is 1024 MiB.

Traces larger than the configured limit are excluded from the analysis. BasicAnalysis reports the traces that have been excluded. If every input trace exceeds the limit, the analysis terminates without computing metrics.

3.10 Debug information

Additional diagnostic information can be enabled with:

```
-d, --debug
```

This option is useful when validating trace detection, metric computation, or report generation.

3.11 Version information

The installed BasicAnalysis version can be displayed with:

```
-v, --version
```

3.12 Command-line help

The complete set of options supported by the installed BasicAnalysis version can always be obtained with:

```
modelfactors.py --help
```

The command-line help should be considered the authoritative reference for the options available in a particular BasicAnalysis version.

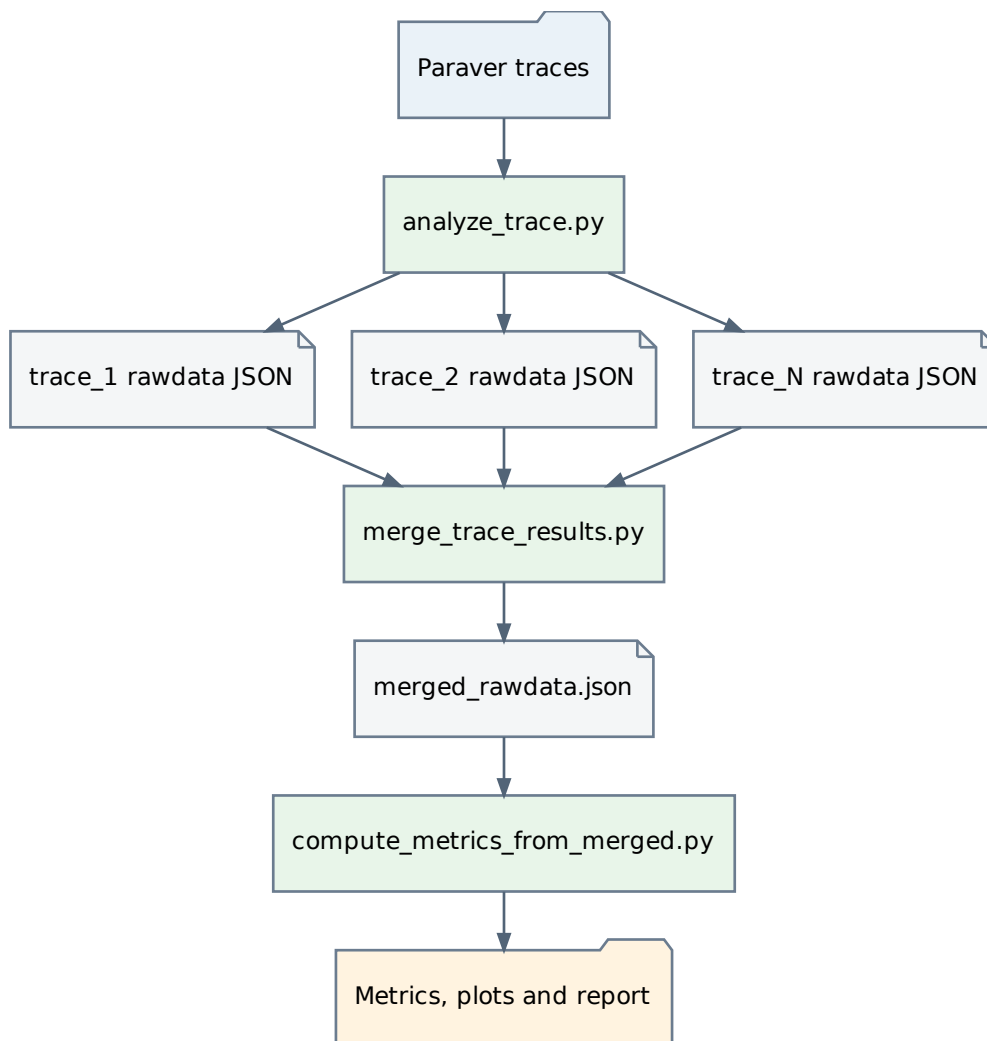
STAGED ANALYSIS WORKFLOW

BasicAnalysis can separate trace processing from metric computation by using a staged analysis workflow.

Instead of analyzing all traces and computing the final metrics in a single `model_factors.py` execution, the analysis can be divided into three independent stages:

1. **Trace analysis:** analyze each Paraver trace and generate a raw-data JSON file.
2. **Raw-data merge:** combine the per-trace raw-data files into a single merged dataset.
3. **Metric computation:** compute the efficiency metrics and other performance indicators, tables, plots, and performance report.

The workflow can be summarized as:



The staged workflow is particularly useful when analyzing multiple large or computationally expensive traces. Because each trace is analyzed independently, trace processing can be distributed across different jobs or compute nodes. This is especially useful when trace processing or simulation requires significant memory or execution time.

The independent per-trace results also make the analysis more flexible. Once a trace has been successfully analyzed, its raw-data file can be preserved and reused in subsequent stages. If a trace needs to be reanalyzed, only that trace has to be processed again, and the resulting raw-data files can then be merged without repeating the other trace analyses.

Finally, separating trace processing from metric computation allows the final metrics and reports to be regenerated from previously extracted raw data without processing the original traces again.

4.1 Trace analysis

The first stage is performed with `analyze_trace.py`:

```
analyze_trace.py [options] [trace_list ...]
```

The command accepts Paraver traces in both uncompressed `.prv` format and compressed `.prv.gz` format.

For example:

```
analyze_trace.py trace_1.prv trace_2.prv trace_3.prv
```

Compressed traces can be analyzed directly:

```
analyze_trace.py trace_1.prv.gz trace_2.prv.gz
```

Wildcard expressions can also be used:

```
analyze_trace.py *.prv
```

The trace-analysis stage performs the operations required to prepare the raw performance information for each execution. Depending on the selected analysis model, this includes:

- validating the input trace;
- detecting the programming model and trace mode;
- extracting performance data from the trace; and
- running the required Dimemas simulations when applicable.

Each successfully analyzed trace produces an independent raw-data JSON file. These files contain the information required by the later metric-computation stage, so the original trace does not need to be analyzed again when only the final metrics or reports need to be regenerated.

4.1.1 Analysis configuration

The trace-analysis stage supports the analysis options that affect raw-data extraction and simulation, including trace-mode detection, simulation configuration, trace-size limits, and parallel processing of independent traces.

For example:

```
analyze_trace.py --jobs auto *.prv
```

can analyze independent traces concurrently.

The main analysis options are described in [Running BasicAnalysis](#). The complete set of options supported by the installed version can be obtained with:

```
analyze_trace.py --help
```

4.2 Merging raw data

After the required traces have been analyzed, their raw-data JSON files can be combined with `merge_trace_results.py`:

```
merge_trace_results.py --output merged_rawdata.json <rawdata-files>
```

For example:

```
merge_trace_results.py --output merged_rawdata.json *.rawdata.json
```

By default, the merge stage orders traces according to the number of processes. This behavior can be controlled with the `-ord` option.

The merge operation combines the independent trace-analysis results into a single dataset that can be used for comparative and scalability analysis.

The merged file preserves the information required for metric computation, including the analyzed trace set, trace metadata, raw performance data, and parallel-configuration information.

Raw-data files do not need to have been generated in the same `analyze_trace.py` execution. This makes it possible to analyze traces independently or to replace the raw data corresponding to a trace that needed to be reanalyzed.

The available command-line options can be displayed with:

```
merge_trace_results.py --help
```

4.3 Computing metrics from merged data

The final stage computes the performance metrics and generates the analysis results from the merged raw-data file:

```
compute_metrics_from_merged.py --merged-input merged_rawdata.json [options]
```

For example:

```
compute_metrics_from_merged.py --merged-input merged_rawdata.json
```

This stage does not analyze the original Paraver traces. It consumes the previously extracted raw data and computes the final efficiency metrics, tables, plots, and interactive performance report.

This separation is useful when the metric configuration or report generation needs to be repeated without rerunning the more expensive trace-analysis stage.

4.3.1 Metric configuration

Options that affect final metric computation can be specified at this stage. These include the metric workflow, scaling model, and the performance model used for MPI+GPU applications.

For example:

```
compute_metrics_from_merged.py \  
  --merged-input merged_rawdata.json \  
  --scaling strong
```

or:

```
compute_metrics_from_merged.py \  
  --merged-input merged_rawdata.json \  
  -pop-model classic
```

The complete set of options can be displayed with:

```
compute_metrics_from_merged.py --help
```

4.4 Reanalyzing failed traces

One of the main advantages of the staged workflow is that an unsuccessful trace analysis does not require all traces to be processed again.

For example, suppose three traces are analyzed:

```
analyze_trace.py trace_1.prv trace_2.prv trace_3.prv
```

and the analysis succeeds for `trace_1` and `trace_3` but fails for `trace_2`.

After resolving the problem affecting `trace_2`, only that trace needs to be analyzed again:

```
analyze_trace.py trace_2.prv
```

The available raw-data files can then be merged:

```
merge_trace_results.py --output merged_rawdata.json *.rawdata.json
```

and the final metrics regenerated:

```
compute_metrics_from_merged.py --merged-input merged_rawdata.json
```

The successful analyses of `trace_1` and `trace_3` therefore do not need to be repeated.

4.5 Complete example

A complete staged analysis can be performed as follows.

First, analyze the traces:

```
analyze_trace.py trace_1.prv trace_2.prv trace_3.prv
```

Second, merge the generated raw-data files:

```
merge_trace_results.py --output merged_rawdata.json *.rawdata.json
```

Finally, compute the performance metrics and generate the reports:

```
compute_metrics_from_merged.py --merged-input merged_rawdata.json
```

The resulting analysis follows the same performance-analysis methodology as the standard BasicAnalysis workflow. The difference is that raw-data extraction and final metric computation are performed as independent stages.

4.6 Choosing between the workflows

The standard workflow with `modelfactors.py` is convenient for direct analysis when all traces can be processed in a single execution.

The staged workflow is preferable when:

- a large number of traces must be analyzed;
- individual trace analyses are expensive;
- traces are processed at different times or on different systems;
- failed traces need to be reanalyzed independently; or
- metrics and reports need to be regenerated without repeating trace processing.

Both workflows ultimately use the same BasicAnalysis metric models and performance-analysis methodology.

PERFORMANCE ANALYSIS METHODOLOGY

BasicAnalysis organizes performance analysis through a hierarchy of efficiency metrics and a set of complementary analytical views. The objective is not only to report individual efficiency values, but also to help users understand which performance factors contribute to the observed efficiency losses and which aspects of the execution should be investigated further.

The analysis can progress from application-level behavior toward more specific performance factors, parallel runtimes, and execution components. Depending on the analysis objective, users can combine several analytical views to understand the overall performance behavior or focus on the view most relevant to a particular performance aspect.

The methodology is organized around the following perspectives:

- **Hierarchical Efficiency Analysis**, which decomposes high-level efficiency metrics into progressively more specific contributing factors.
- **Parallel Runtime Model**, which attributes parallel-efficiency behavior to the active parallel runtimes.
- **Runtime-Specific Analysis**, which provides additional metrics describing the behavior of individual parallel runtimes.
- **Execution Domains**, which provides a complementary Host/Device analysis for accelerator applications.
- **I/O Analysis**, which provides complementary metrics characterizing the contribution and distribution of File I/O activity.
- **Scaling Analysis**, which examines how performance and efficiency factors evolve across execution configurations.

The following sections describe how these perspectives are related and how they can be used according to the objective of the performance assessment.

5.1 Hierarchical efficiency analysis

As BasicAnalysis automates the computation of POP performance metrics, it follows the POP hierarchical metric model, in which high-level efficiencies are decomposed into more specific contributing factors. This hierarchy allows the analysis to progress from an overall efficiency loss toward increasingly specific factors that can explain it.

At the application level, the hierarchy starts with Global Efficiency, which is decomposed into Parallel Efficiency and Computation Scalability:

$$Global\ Efficiency = Parallel\ Efficiency \times Computation\ Scalability$$

These two factors represent the two main sources of inefficiency in parallel applications. Parallel Efficiency captures the overheads introduced by parallel execution, while Computation Scalability captures how the computation scales as the computational resources increase.

5.1.1 Parallel Efficiency

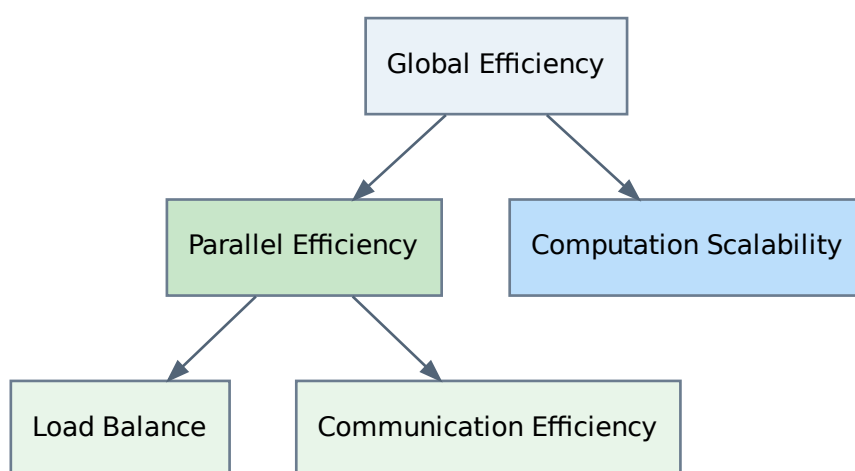
Parallelization requires dividing the application workload among the available parallel units and defining the interactions required among those units to complete the computation.

Parallel Efficiency characterizes the efficiency of this parallelization. Its losses can arise from two fundamental aspects: an uneven distribution of the computational workload among the parallel units and the overhead associated with the communication and coordination required among them.

Therefore, Parallel Efficiency is decomposed as:

$$\text{Parallel Efficiency} = \text{Load Balance} \times \text{Communication Efficiency}$$

This results in the following application-level hierarchy:



Load Balance characterizes how evenly the computational workload is distributed among the parallel units, while Communication Efficiency characterizes the efficiency loss associated with the communication and coordination required among them.

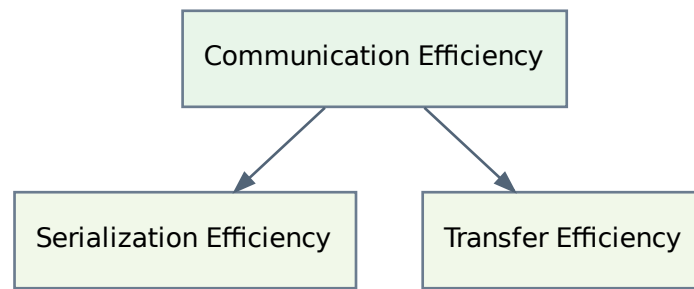
At this level of the hierarchy, communication is a general parallel-execution concept and is not restricted to message passing or to a particular programming model. Its specific interpretation depends on the parallel execution model being analyzed.

Communication Efficiency can be further decomposed to distinguish losses associated with serialization from those associated with data transfer:

$$\text{Communication Efficiency} = \text{Serialization Efficiency} \times \text{Transfer Efficiency}$$

Serialization Efficiency characterizes losses associated with dependencies and synchronization among parallel units, while Transfer Efficiency characterizes the additional cost associated with transferring the information required for their interaction.

The Communication Efficiency branch can therefore be represented as:



The way these communication factors are obtained depends on the parallel execution model. BasicAnalysis directly derives the MPI factors using idealized communication information generated through Dimemas. For hybrid executions, additional runtime contributions can be derived from the relationships between the hybrid- and runtime-level metrics, as described in the Runtime-Specific Analysis section.

5.1.2 Computation Scalability

The other main factor contributing to Global Efficiency is Computation Scalability. While Parallel Efficiency captures the overheads introduced by parallel execution, Computation Scalability characterizes how the computation scales as the computational resources increase.

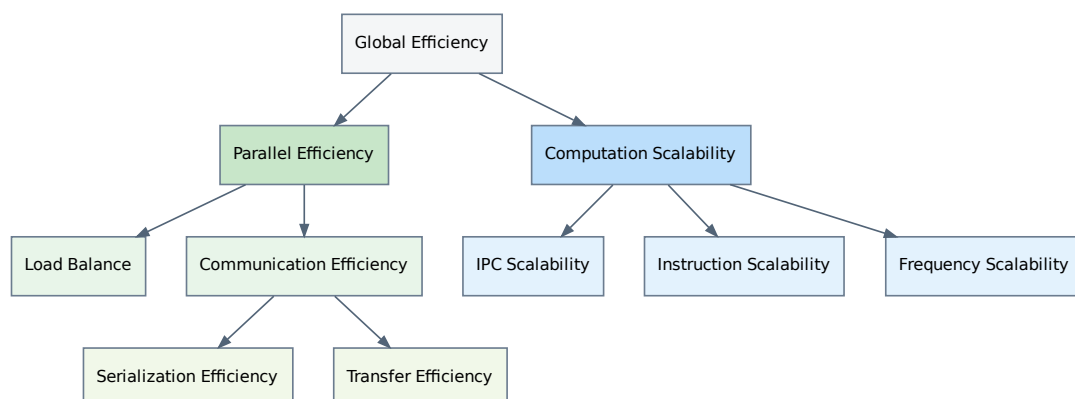
Unlike Parallel Efficiency, Computation Scalability is evaluated relative to a reference execution. It captures changes in the computational behavior as the application scales, independently of the overheads attributed to parallel execution.

Computation Scalability is decomposed as:

$$\text{Computation Scalability} = \text{IPC Scalability} \times \text{Instruction Scalability} \times \text{Frequency Scalability}$$

The scaling of the computation can therefore be decomposed into three components: Instructions, IPC, and Frequency. These represent the three main factors that determine the duration of the computation: the amount of work performed (Instructions), the rate at which that work is executed (IPC), and the operating speed of the computational resource (Frequency).

Together with the Parallel Efficiency decomposition introduced above, these metrics complete the application-level efficiency hierarchy:



The hierarchy provides a systematic way to navigate an efficiency analysis. When a parent metric shows a significant loss, its child metrics can be inspected to determine which performance factor contributes to that loss.

For example, a low Parallel Efficiency can lead to the inspection of Load Balance and Communication Efficiency. If Communication Efficiency shows the larger loss, Serialization Efficiency and Transfer Efficiency can then be examined to distinguish whether the degradation is primarily associated with dependencies and synchronization or with data-transfer costs.

Similarly, Computation Scalability can be investigated through its IPC, Instruction, and Frequency Scalability components to determine which factor contributes to the observed degradation.

The analysis can continue through the hierarchy until the available metrics provide the level of detail required for the performance assessment.

This application-level hierarchy provides the basis for the additional analysis performed by BasicAnalysis. When multiple parallel runtimes participate in the execution, the Parallel Efficiency branch can be extended to distinguish the contribution of each runtime while preserving the same underlying performance factors. This extension is introduced by the Parallel Runtime Model in the following section.

The complete definitions and formulations of these metrics are provided in [Performance Metrics](#).

5.2 Parallel Runtime Model

The application-level hierarchy introduced in the previous section can also be applied to applications that combine multiple parallel runtimes. In this case, however, Parallel Efficiency and its contributing factors characterize the combined behavior of the parallel runtimes participating in the execution.

For a hybrid application, this application-level Parallel Efficiency is referred to as **Hybrid Parallel Efficiency**. Similarly, Hybrid Load Balance and Hybrid Communication Efficiency characterize workload distribution and communication and coordination for the hybrid execution as a whole.

This decomposition identifies which performance factor limits the hybrid execution, but it does not reveal how the different parallel runtimes contribute to that factor. To provide this additional level of analysis, BasicAnalysis uses the **Parallel Runtime Model**.

The Parallel Runtime Model introduces a runtime dimension into the Parallel Efficiency hierarchy. For a hierarchical hybrid execution, it distinguishes an **outer parallel runtime** and an **inner parallel runtime**. For example, MPI is the outer runtime and OpenMP the inner runtime in MPI+OpenMP, while MPI is the outer runtime and the accelerator runtime the inner runtime in MPI+CUDA and MPI+HIP.

5.2.1 Parallel-runtime performance factors

The Parallel Runtime Model preserves the three performance factors introduced for parallel execution—Parallel Efficiency, Load Balance, and Communication Efficiency—and evaluates them at different runtime levels.

For compactness, the following notation is used in the equations in this section:

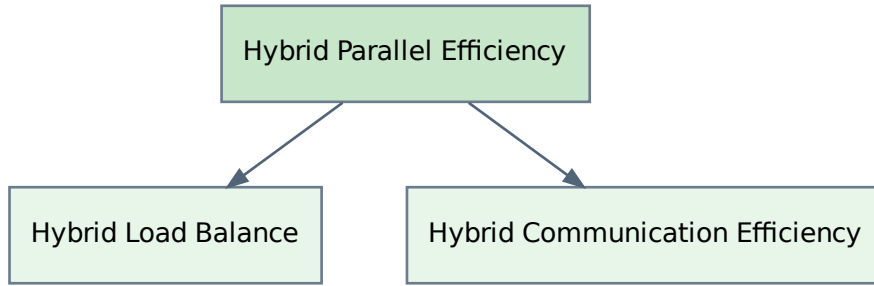
- **PE**: Parallel Efficiency
- **LB**: Load Balance
- **CommE**: Communication Efficiency

The model distinguishes three levels: the **hybrid level**, the **outer-runtime level**, and the **inner-runtime level**.

At the hybrid level, the metrics characterize the complete hybrid parallelization, considering all participating parallel runtimes. The performance-factor decomposition remains:

$$Hybrid_PE = Hybrid_LB \times Hybrid_CommE$$

Here, *Hybrid_LB* characterizes workload distribution across the complete hybrid execution, while *Hybrid_CommE* characterizes the communication and coordination overhead associated with all participating parallel runtimes.

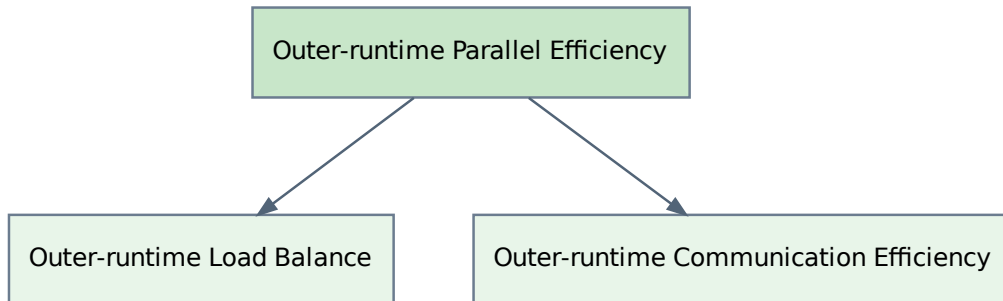


To isolate the contribution of the outer runtime, the execution is analyzed from its perspective. Activity associated with the outer runtime is treated as parallel-runtime overhead, while execution outside that runtime is considered computation at this level.

The same performance-factor decomposition is preserved:

$$Outer_PE = Outer_LB \times Outer_CommE$$

For MPI+OpenMP, MPI is the outer runtime. From the MPI perspective, time inside MPI represents MPI-runtime activity, while execution outside MPI, including OpenMP activity, is considered computation. The resulting MPI metrics therefore characterize the performance factors associated with the outer parallelization.



Once the hybrid and outer-runtime factors are known, the contribution of the inner runtime can be derived using the multiplicative structure of the model. For Parallel Efficiency:

$$Hybrid_PE = Outer_PE \times Inner_PE$$

The same relationship applies to Load Balance and Communication Efficiency:

$$Hybrid_LB = Outer_LB \times Inner_LB$$

$$Hybrid_CommE = Outer_CommE \times Inner_CommE$$

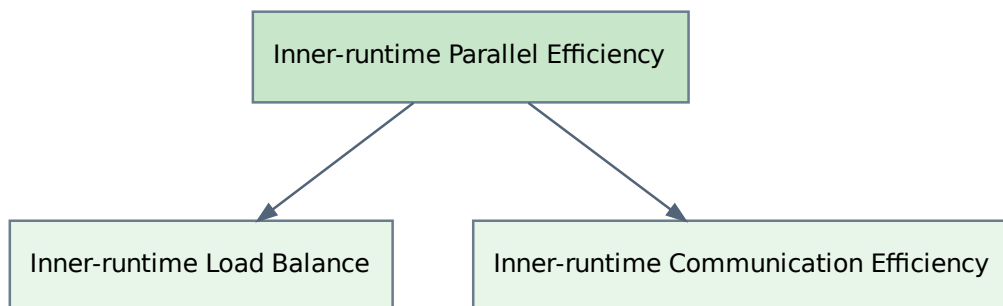
Therefore, the inner-runtime performance factors are obtained as:

$$Inner_PE = \frac{Hybrid_PE}{Outer_PE}$$

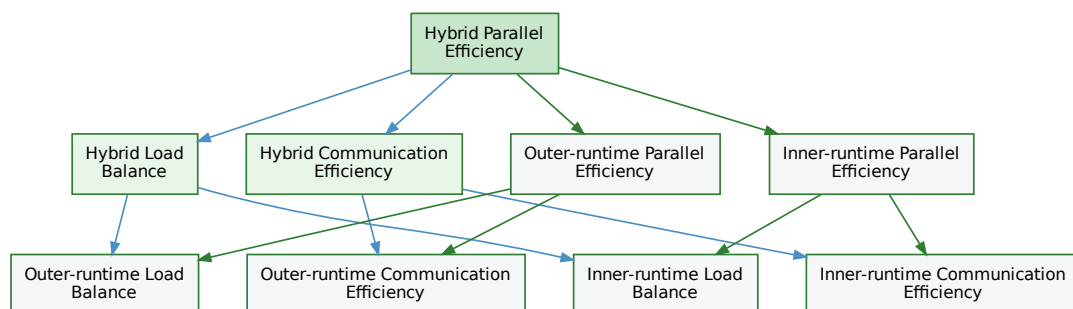
$$Inner_LB = \frac{Hybrid_LB}{Outer_LB}$$

$$Inner_CommE = \frac{Hybrid_CommE}{Outer_CommE}$$

The inner-runtime metrics therefore represent the contribution required to relate the outer-runtime factors to those observed for the complete hybrid execution.



The resulting generic metric hierarchy for a hybrid parallel application can be represented as follows:



The hierarchy can be followed either by performance factor, to examine Load Balance or Communication Efficiency across runtime levels, or by runtime, to examine the factors contributing to the parallel efficiency of the outer or inner runtime.

For an MPI+OpenMP execution, the outer and inner runtimes correspond to MPI and OpenMP, respectively. The generic runtime decomposition described above can therefore be expressed directly in terms of these two runtimes.

For Parallel Efficiency, the relationship becomes:

$$Hybrid_PE = MPI_PE \times OpenMP_PE$$

and the OpenMP contribution is therefore obtained as:

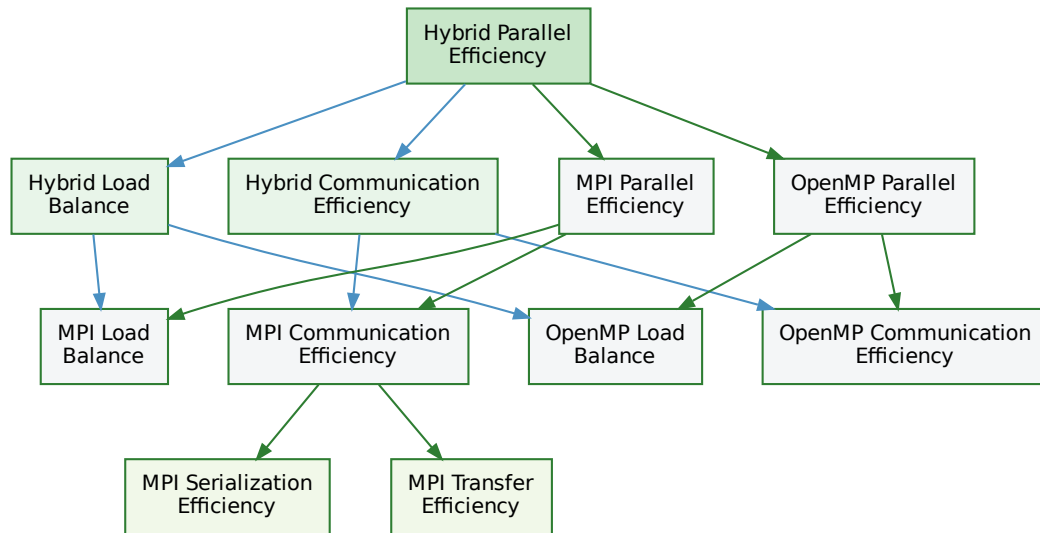
$$OpenMP_PE = \frac{Hybrid_PE}{MPI_PE}$$

The same decomposition applies to the other two performance factors. Hybrid Load Balance is determined by the MPI and OpenMP Load Balance contributions, while Hybrid Communication Efficiency is determined by the corresponding MPI and OpenMP Communication Efficiency contributions:

$$Hybrid_LB = MPI_LB \times OpenMP_LB$$

$$Hybrid_CommE = MPI_CommE \times OpenMP_CommE$$

The resulting Parallel Runtime Model for an MPI+OpenMP application can therefore be represented as:



When MPI is the outer runtime, MPI Communication Efficiency can be further decomposed into MPI Serialization Efficiency and MPI Transfer Efficiency. These factors provide a deeper characterization of the communication losses associated with the MPI runtime.

5.2.2 Interpreting derived runtime contributions

Unlike the hybrid and outer-runtime factors, inner-runtime contributions are not independent efficiency measurements. They are derived from the multiplicative relationship between the hybrid and outer-runtime levels.

Because an inner-runtime contribution is obtained as the ratio between the corresponding hybrid and outer-runtime factors, its value may exceed 100% when the hybrid factor is greater than the outer-runtime factor. Such a value should not be interpreted as an independently measured physical efficiency above its ideal value, but as the relative contribution of the inner runtime within the multiplicative model.

Inner-runtime factors should therefore be interpreted together with the hybrid and outer-runtime factors from which they are derived.

5.3 Runtime-Specific Analysis

The Parallel Runtime Model identifies how the active parallel runtimes contribute to the main parallel-performance factors. Once a runtime is identified as contributing to an efficiency loss, however, additional information may be required to understand the behavior behind that contribution.

BasicAnalysis therefore complements the Parallel Runtime Model with **Runtime-Specific Analysis**. Depending on the programming model, this analysis can either extend the decomposition of the common performance factors or introduce additional metrics that characterize behavior specific to a particular runtime.

These two perspectives are complementary. The Parallel Runtime Model preserves common performance factors across runtime levels, allowing their contributions to the hybrid execution to be distinguished. Runtime-specific metrics provide a more specialized view of the execution and can expose sources of inefficiency that are meaningful for a particular programming model.

The runtime-specific information currently available depends on the parallel programming model and on the information that can be obtained from the analyzed traces.

5.3.1 Extending the Parallel Runtime Model

The Parallel Runtime Model introduced in the previous section attributes Parallel Efficiency, Load Balance, and Communication Efficiency to the parallel runtimes participating in a hybrid execution. When additional information is available, the same multiplicative principle can be extended to the factors explaining Communication Efficiency.

Communication Efficiency can be decomposed into Serialization Efficiency and Transfer Efficiency:

$$CommE = Serialization \times Transfer$$

Serialization Efficiency characterizes losses associated with dependencies and synchronization among parallel units, while Transfer Efficiency characterizes the additional overhead associated with the mechanisms required to perform their interaction.

For an MPI+OpenMP execution, BasicAnalysis extends this decomposition to the hybrid and runtime levels. This makes it possible not only to identify whether Communication Efficiency is limiting the hybrid execution, but also to distinguish how MPI and OpenMP contribute to its Serialization and Transfer components.

Hybrid and MPI communication factors

The MPI and hybrid communication factors are characterized from different execution perspectives.

At the MPI level, BasicAnalysis uses an idealized MPI execution to separate the effects associated with serialization from those associated with data transfer. OpenMP activity is not modeled as an independent source of parallel-runtime overhead at this level.

At the hybrid level, both MPI and OpenMP participate in the execution model. Idealized execution information is used to separate structural limitations from the additional overheads introduced by the mechanisms used to coordinate and communicate among the parallel units.

This provides Serialization and Transfer factors at both the hybrid and MPI levels. The remaining contribution can then be attributed to the OpenMP runtime using the same multiplicative principle introduced by the Parallel Runtime Model.

Deriving the OpenMP contribution

Following the same multiplicative approach used for the other parallel-runtime factors, Hybrid Serialization Efficiency is expressed in terms of the MPI and OpenMP contributions:

$$Hybrid_Serialization = MPI_Serialization \times OpenMP_Serialization$$

Similarly, Hybrid Transfer Efficiency is expressed as:

$$Hybrid_Transfer = MPI_Transfer \times OpenMP_Transfer$$

The OpenMP contributions can therefore be derived from the corresponding hybrid and MPI factors:

$$OpenMP_Serialization = \frac{Hybrid_Serialization}{MPI_Serialization}$$
$$OpenMP_Transfer = \frac{Hybrid_Transfer}{MPI_Transfer}$$

These relationships extend the runtime attribution introduced for Load Balance and Communication Efficiency to the two factors that explain Communication Efficiency.

Interpreting the OpenMP communication factors

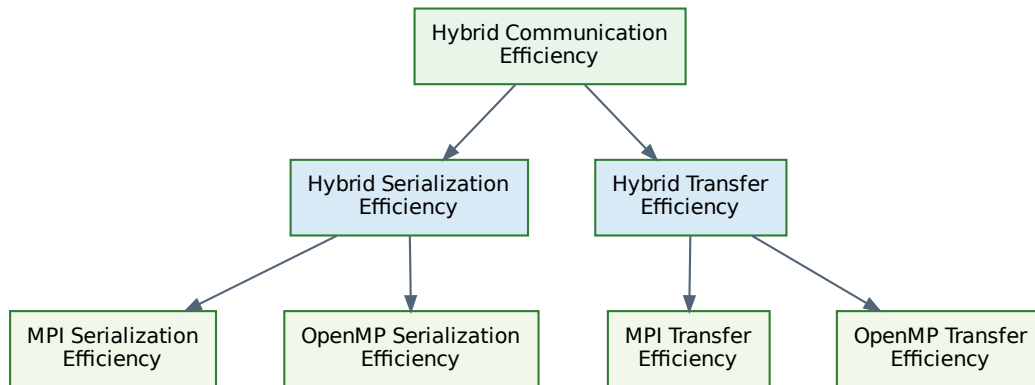
At the OpenMP level, Serialization and Transfer Efficiency should be interpreted according to the mechanisms through which threads interact rather than in terms of message passing.

OpenMP Serialization Efficiency characterizes structural limitations that restrict concurrent execution, such as dependencies or limited available parallelism.

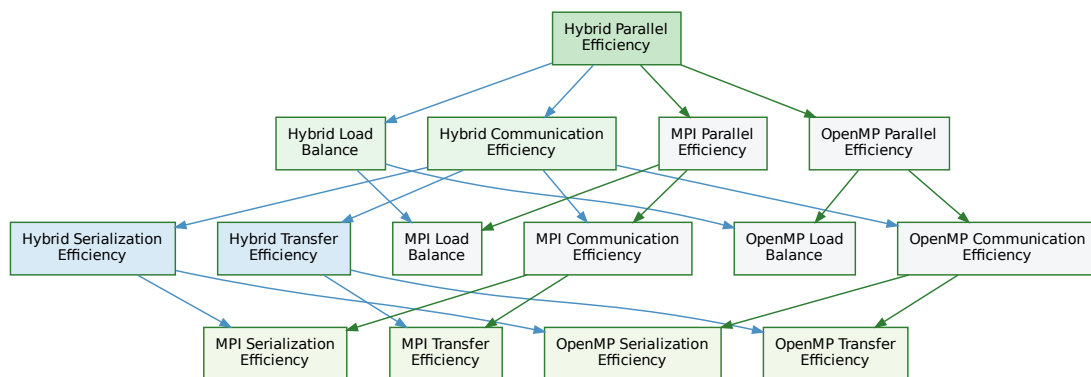
OpenMP Transfer Efficiency characterizes additional OpenMP runtime overheads associated with coordinating the parallel execution, including synchronization, scheduling, and thread-management activity.

The terminology therefore preserves the common Communication Efficiency decomposition while its runtime-specific interpretation reflects the interaction mechanisms of OpenMP.

The resulting hierarchy can be represented as:



The OpenMP factors derived through this extension describe the contribution of OpenMP to the communication-related factors observed in the hybrid execution. They remain relative contributions within the Parallel Runtime Model. The MPI+OpenMP metric hierarchy is therefore extended as follows:



A different perspective is required to characterize the OpenMP execution itself. For this purpose, BasicAnalysis provides an isolated OpenMP analysis that evaluates serial execution, load imbalance within parallel regions, and OpenMP scheduling and fork/join overhead.

5.3.2 Isolated OpenMP analysis

The OpenMP contributions derived through the Parallel Runtime Model describe how the inner runtime contributes to the performance factors observed for the complete hybrid execution. BasicAnalysis also provides an isolated OpenMP analysis that examines the efficiency of the OpenMP execution itself.

Efficient OpenMP execution requires more than distributing work among threads. The application must expose sufficient parallel work, distribute that work evenly among the threads participating in each parallel region, and manage the parallel execution without excessive runtime overhead.

The isolated OpenMP model therefore considers three main sources of inefficiency: serial execution outside

OpenMP parallel regions, load imbalance among threads inside parallel regions, and OpenMP runtime overhead associated with scheduling and fork/join activity.

Serial execution

Execution outside OpenMP parallel regions limits the amount of work that can be performed concurrently by the available threads. During these regions, only the master thread is active while the remaining OpenMP threads do not participate in the computation.

OpenMP Serial Efficiency characterizes the efficiency loss associated with this serial part of the OpenMP execution. A low value therefore indicates that a significant fraction of the available thread execution capacity is lost because execution remains outside OpenMP parallel regions.

Load imbalance in parallel regions

Entering an OpenMP parallel region does not by itself guarantee efficient use of the participating threads. The useful computational work must also be distributed evenly among them.

OpenMP Load Balance characterizes differences in useful computation among threads within each OpenMP parallel region. When some threads perform less useful work than others, they finish their assigned work earlier and remain without useful computation while other threads are still active.

A low OpenMP Load Balance therefore indicates an uneven distribution of useful work inside OpenMP parallel regions.

Scheduling and fork/join overhead

Even when sufficient parallel work is available and that work is well balanced, the OpenMP runtime introduces overhead to create, coordinate, and schedule the parallel execution.

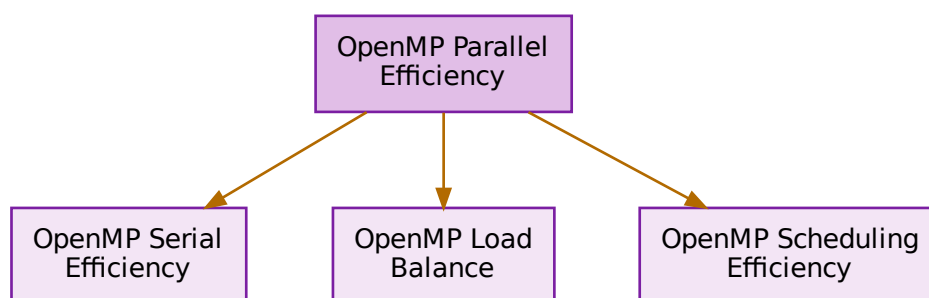
OpenMP Scheduling Efficiency characterizes runtime overhead associated with OpenMP scheduling and fork/join activity that is not attributed to serial execution or useful-work imbalance.

A low OpenMP Scheduling Efficiency therefore indicates that OpenMP runtime management represents a significant source of efficiency loss.

These three factors determine the isolated OpenMP Parallel Efficiency:

$$OMP_PE = OMP_Serial \times OMP_LB \times OMP_Sched$$

The metric hierarchy for the isolated OpenMP analysis can therefore be represented as:



5.4 Execution Domains

Accelerator applications involve execution across two distinct but interacting domains: the **Host**, where the CPU-side execution takes place and work is prepared and submitted to the accelerator, and the **Device**, where the offloaded work is executed.

This distinction is independent of the particular parallel programming model. The Host may itself use a distributed- or shared-memory parallel runtime, while the Device execution is managed through an accelerator programming model. For example, the same Host/Device perspective can conceptually be applied to MPI+GPU, OpenMP+GPU, or OpenACC+GPU applications.

The performance behavior observed in these two domains is complementary but different. Inefficiency may originate from the Host execution or from the mechanisms used to supply work to the accelerator, while other losses may manifest on the Device through insufficient available work, data movement, workload imbalance, or computation scalability.

The **Execution Domains** analysis separates these two perspectives in order to identify where accelerator-related inefficiencies manifest:

- **Host**, characterizing CPU-side execution and the interaction with the accelerator;
- **Device**, characterizing the execution of work on the accelerator.

BasicAnalysis currently implements this analysis for MPI+GPU applications, where MPI represents the Host-side parallel runtime and CUDA or HIP manages the accelerator execution.

The Host/Device efficiency model extends the POP methodology to heterogeneous accelerated systems. BasicAnalysis applies this model through post-mortem analysis of Paraver traces, deriving the Host- and Device-side execution information required to compute the corresponding metrics.

For compactness, the following notation is used for the Host and Device metrics:

- **Host_PE**: Host Parallel Efficiency
- **MPI_PE**: MPI Parallel Efficiency
- **Device_Offload**: Device Offload Efficiency
- **Host_GE**: Host Global Efficiency
- **Host_CompScale**: Host Computation Scalability
- **Device_GE**: Device Global Efficiency
- **Device_PE**: Device Parallel Efficiency
- **Device_LB**: Device Load Balance
- **Device_CommE**: Device Communication Efficiency
- **Device_OrchE**: Device Orchestration Efficiency
- **Device_CompScale**: Device Computation Scalability

5.4.1 Host Global Efficiency

The Host Global Efficiency characterizes the efficiency of the CPU-side execution. Performance losses at this level can arise from the parallel execution and the interaction with the accelerator, but also from changes in the efficiency of the computation performed on the Host as the execution configuration changes.

These two aspects are characterized by Host Parallel Efficiency and Host Computation Scalability, respectively. Host Global Efficiency is therefore defined as:

$$Host_GE = Host_PE \times Host_CompScale$$

Host Parallel Efficiency

From the Host perspective, parallel-efficiency losses can originate from two different components of the execution. The Host parallel runtime can introduce the usual load-distribution and communication overheads, while interaction with the accelerator can introduce offload overhead associated with launching work, transferring data, or waiting for the device.

In the current MPI+GPU implementation, the Host-side parallel runtime is MPI. Therefore, Host Parallel Efficiency is decomposed into an MPI contribution and an accelerator-offload contribution:

$$Host_PE = MPI_PE \times Device_Offload$$

MPI Parallel Efficiency characterizes the contribution associated with the MPI parallelization. When evaluating this contribution, time spent outside MPI, including Host activity associated with managing the accelerator, is considered computation from the MPI perspective.

Device Offload Efficiency characterizes the remaining Host-side loss associated with using the accelerator. This includes activity such as launching accelerator work, transferring data, and waiting for accelerator operations to complete.

Host Computation Scalability

Host Computation Scalability characterizes how the useful computation performed on the Host evolves when the execution configuration changes. It applies the computation-scalability branch of the performance model to Host-side useful computation.

As in the application-level model, Host Computation Scalability can be further decomposed into:

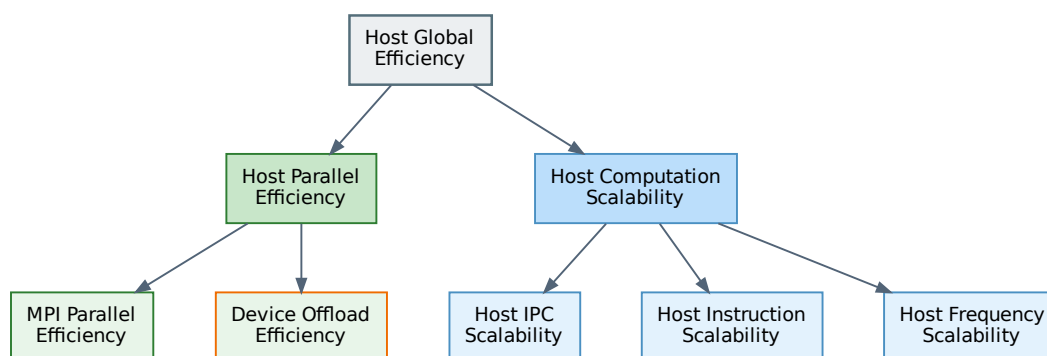
- **Host IPC Scalability**, characterizing changes in the number of instructions completed per processor cycle;
- **Host Instruction Scalability**, characterizing changes in the amount of instructions required to perform the useful Host computation;
- **Host Frequency Scalability**, characterizing changes in processor frequency.

These factors are evaluated considering only the Host-side execution and therefore characterize the scalability of the CPU-side computation independently of the Device execution.

The Host computation-scaling branch is therefore decomposed into:

$$Host_CompScale = Host_IPCScale \times Host_InstructionScale \times Host_FrequencyScale$$

Together, the Host Parallel Efficiency and Host Computation Scalability branches provide the complete decomposition of Host Global Efficiency. The resulting Host metric hierarchy for the current MPI+GPU implementation is:



5.4.2 Device Global Efficiency

Device Global Efficiency characterizes the efficiency of the accelerator-side execution. Performance losses at this level can arise from how effectively the available Device resources execute the work supplied by the Host, but also from changes in the efficiency of the computation performed on the Device as the execution configuration changes.

These two aspects are characterized by Device Parallel Efficiency and Device Computation Scalability, respectively. Device Global Efficiency is therefore defined as:

$$Device_GE = Device_PE \times Device_CompScale$$

Device Parallel Efficiency

A device may fail to perform useful computation for several different reasons. It may not have work available to execute, it may be involved in data movement, or the useful work may be distributed unevenly across the available devices. The Device hierarchy separates these effects into three performance factors: Orchestration Efficiency, Communication Efficiency, and Load Balance.

Device Load Balance

When several accelerators participate in the execution, useful work must also be distributed evenly among them.

Device Load Balance characterizes differences in useful computation across devices. A low value indicates that some devices perform less useful computation than others and therefore contribute less effectively to the parallel execution.

Device Communication Efficiency Even when work is available, device execution can be delayed by data movement. Transfers may occur between Host and Device or between accelerators, and may also involve communication mechanisms such as accelerator-aware MPI or device-communication libraries.

Device Communication Efficiency characterizes the loss associated with data movement that is not hidden by computation. Data transfers that overlap with useful computation do not contribute to this loss.

Device Orchestration Efficiency

A device can remain idle because executable work is not available when it is ready to execute. This may reflect inefficiencies in coordinating computation, communication, and Host offload, including delays in supplying work or dependencies between accelerator operations.

Device Orchestration Efficiency characterizes the loss associated with periods in which the accelerator remains inactive because neither useful computation nor data movement is being performed. A low value therefore indicates that the accelerator is not being supplied with executable work continuously.

Together, these factors determine Device Parallel Efficiency:

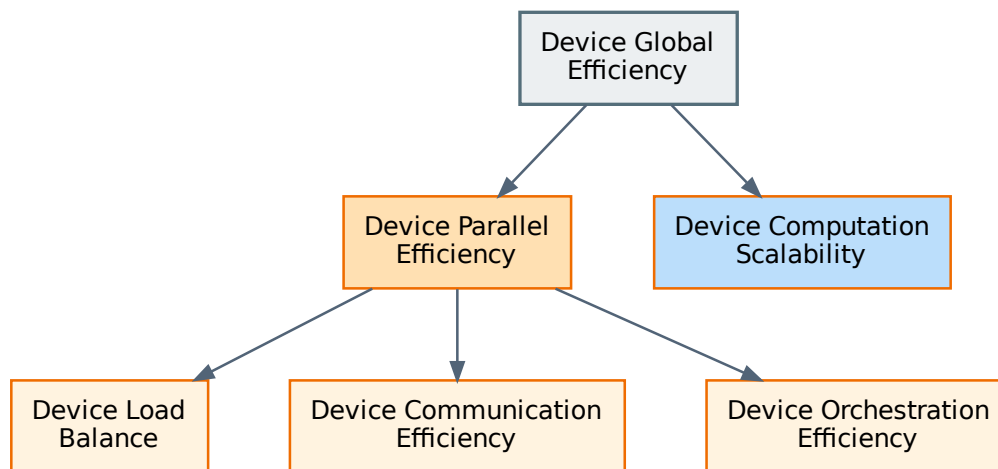
$$Device_PE = Device_LB \times Device_CommE \times Device_OrchE$$

Device Computation Scalability

Device Computation Scalability characterizes how the useful computation performed on the Device evolves when the execution configuration changes. It complements Device Parallel Efficiency by separating changes in the computation itself from losses associated with workload distribution, data movement, and the availability of work on the accelerator.

At present, BasicAnalysis provides Device Computation Scalability as the parent metric of the computation-scaling branch. A decomposition into lower-level Device computation-scaling factors is not currently defined in BasicAnalysis and is therefore not included in the metric hierarchy.

With the Device Parallel Efficiency factors described above and Device Computation Scalability completing the second branch, the Device Global Efficiency hierarchy for the current MPI+GPU implementation can be represented as:



5.4.3 Independent Host and Device efficiencies

The Host and Device hierarchies describe different execution domains. They are not successive levels of a single multiplicative hierarchy.

BasicAnalysis therefore computes two independent domain-level efficiencies:

- **Host Global Efficiency**, which characterizes CPU-side execution and the interaction with the accelerator;
- **Device Global Efficiency**, which characterizes execution on the accelerator itself.

A low Host Global Efficiency may indicate limitations in the CPU-side execution or offload path, while a low Device Global Efficiency may indicate insufficient work, data-movement overhead, imbalance, or poor computation scalability on the accelerator.

Warning

Host Global Efficiency and Device Global Efficiency must not be multiplied to obtain application Global Efficiency.

The Execution Domains analysis should therefore be interpreted as a localization view: it helps determine **where** accelerator-related inefficiency manifests. The Parallel Runtime Model provides a different, complementary perspective by attributing efficiency losses to the participating parallel runtimes and their common performance factors.

5.5 I/O analysis

File I/O can represent an important component of application execution, but its contribution is not currently part of the hierarchical efficiency model used by BasicAnalysis. BasicAnalysis therefore provides **I/O Analysis** as a complementary analytical view.

The analysis characterizes two aspects of the observed File I/O activity:

- the relative contribution of File I/O to the measured execution activity;
- the distribution of File I/O activity across the execution units involved.

BasicAnalysis distinguishes File I/O performed through **MPI-I/O** from **POSIX and ANSI C File I/O** when the corresponding activity is available in the trace. The resulting metrics allow the user to determine whether File

I/O represents a significant component of the execution and whether the observed I/O activity is evenly distributed across the participating execution units.

The I/O metrics are interpreted as efficiency-oriented metrics, so higher values represent more favorable behavior. However, they should not be multiplied with Global Efficiency, Parallel Efficiency, or other factors of the hierarchical performance model.

When several execution configurations are analyzed, the evolution of the I/O metrics can also be compared across configurations to identify changes in the relative contribution or distribution of File I/O activity as the application scales.

The definitions, formulations, and availability conditions of the I/O metrics are provided in *Performance Metrics*.

5.6 Scaling analysis

When several execution configurations are analyzed together, BasicAnalysis examines how application performance and the efficiency factors identified by the hierarchical model evolve as the amount of parallel resources changes.

The **Scaling Analysis** adds the configuration dimension to the performance assessment. While the hierarchical analysis identifies the factors that contribute to an efficiency loss for a given execution, Scaling Analysis examines how those factors evolve across the analyzed configurations.

Scaling Analysis should not be confused with the **Computation Scalability** metric introduced earlier in this chapter. Computation Scalability is one factor of the efficiency hierarchy, characterizing how the computation itself scales relative to a reference execution. Scaling Analysis is the broader comparative analysis in which Computation Scalability, Parallel Efficiency, and their contributing factors can all be examined across configurations.

The interpretation of the scaling behavior depends on the scaling model. BasicAnalysis supports both **strong scaling**, where the problem size remains constant while the computational resources increase, and **weak scaling**, where the amount of work increases together with the computational resources.

5.6.1 Determining the scaling model

When automatic scaling detection is enabled, BasicAnalysis determines whether the analyzed executions exhibit behavior consistent with strong or weak scaling from the execution measurements.

The first execution in the ordered trace list is used as the reference. BasicAnalysis evaluates three indicators across the remaining configurations:

- the growth in useful instructions relative to the growth in the number of processes;
- the evolution of execution time relative to the reference;
- the evolution of average useful computation relative to the reference.

For each indicator, BasicAnalysis computes its average relative behavior across the non-reference executions. An indicator is considered consistent with weak scaling when its average ratio is greater than 0.9.

Weak scaling is selected when at least two of the three indicators exhibit weak-scaling behavior. Otherwise, the executions are classified as strong scaling.

This detection provides an execution-based estimate of the scaling model rather than determining the application problem size directly. The scaling model can therefore be selected explicitly when the intended scaling experiment is known. If the manually selected model differs from the automatically detected one, BasicAnalysis reports the discrepancy.

The detected scaling model, the model finally used for the analysis, and whether the selection was automatic or manual are included in the scaling information reported by BasicAnalysis.

5.6.2 Reference execution

The reference execution used during scaling detection also provides the baseline for evaluating relative scalability metrics, such as Computation Scalability and its contributing factors.

By default, BasicAnalysis orders the traces according to their parallel configuration and uses the first execution in this ordering as the reference. Users can preserve the input ordering when a different reference execution is required.

The choice of reference affects the numerical values of the relative scalability metrics and their interpretation. The selected configuration should therefore provide a meaningful baseline for evaluating how performance evolves as the application scales.

5.6.3 Interpreting scaling trends

Scaling Analysis preserves the relationships defined by the metric hierarchy, but examines them across execution configurations. The evolution of a parent metric can therefore be interpreted together with the evolution of its contributing factors.

For example, if Global Efficiency decreases as the application scales, Parallel Efficiency and Computation Scalability can be compared across the same configurations to determine which branch is responsible for the observed degradation. If Parallel Efficiency deteriorates, its Load Balance and Communication Efficiency trends can then be inspected. Similarly, a degradation in Computation Scalability can be investigated through the evolution of its IPC, Instruction, and Frequency Scalability factors.

This comparative perspective is important because the scaling behavior of a metric is not determined only by its value at a single configuration. A metric may remain relatively high while progressively degrading as resources increase, revealing an emerging scalability limitation. Conversely, a metric that is low but remains stable may represent an existing performance limitation without being the factor responsible for the observed scaling degradation.

5.7 Complementary analytical views

The BasicAnalysis analytical views address different performance questions and provide complementary perspectives on the execution. They are not intended to be interpreted as independent analyses, but as different levels of evidence that can be combined according to the objective of the performance assessment.

A useful way to interpret them is:

Hierarchical Efficiency Analysis

Which performance factor contributes to the observed efficiency loss?

Parallel Runtime Model

When multiple parallel runtimes participate in the execution, which runtime contributes to that performance factor?

Runtime-Specific Analysis

Which behavior within the identified runtime helps explain the observed inefficiency?

Execution Domains

For accelerator applications, where does the inefficiency manifest: Host, offload path, or Device?

I/O Analysis

How significant is File I/O activity, and how is that activity distributed across the execution units involved?

Scaling Analysis

How do the performance factors and their contributions evolve as the execution configuration changes?

These perspectives can be followed progressively during a comprehensive performance assessment or selected individually when investigating a specific performance question. Their combination allows the analysis to move from identifying an efficiency loss, to attributing it to a performance factor or parallel runtime, and, when supported

by the available metrics, to further characterizing where and how the inefficiency manifests. Complementary views can additionally expose execution aspects, such as File I/O activity, that are relevant to the performance assessment without forming part of the hierarchical efficiency model.

BasicAnalysis provides the efficiency metrics and interpretation guidance needed to support this assessment. The metrics identify and characterize performance losses, but they do not necessarily establish their underlying cause. Detailed trace inspection or specialized performance-analysis tools may therefore be required to validate the observations and determine why the identified behavior occurs.

PERFORMANCE METRICS

BasicAnalysis computes a hierarchy of efficiency and scalability metrics from the performance information extracted from the analyzed Paraver traces.

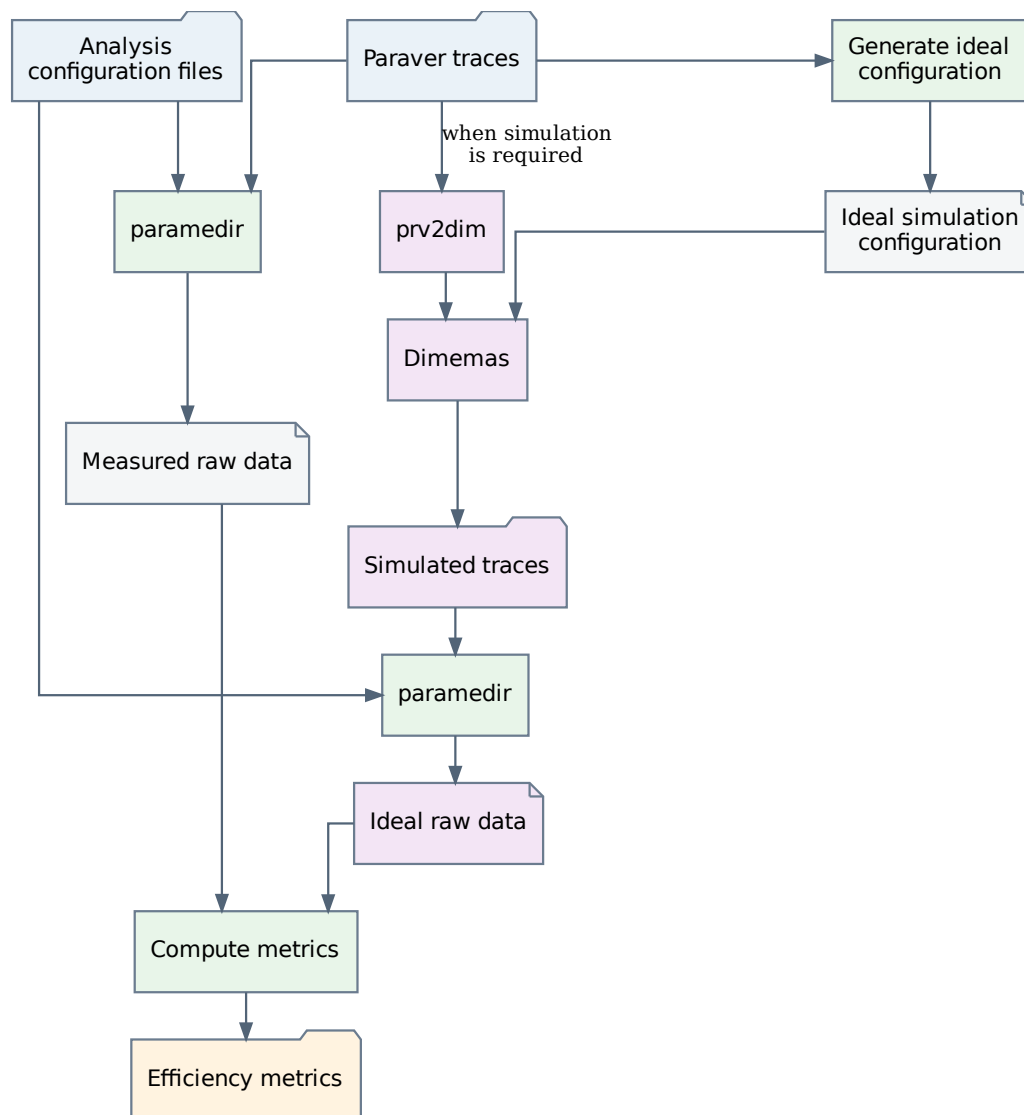
This chapter describes the metrics as they are implemented by BasicAnalysis. For each metric, the formulation corresponds to the quantities actually used by the tool. The conceptual relationships between the metrics and their role in the performance-analysis methodology are described in *Performance Analysis Methodology*.

Metric values are reported as percentages in the efficiency tables and Performance Report unless otherwise indicated.

6.1 From traces to efficiency metrics

BasicAnalysis derives its performance metrics from information extracted from Paraver traces. Depending on the metric and programming model, the required information can come directly from the measured execution or from an idealized execution generated through simulation.

The internal analysis workflow combines Paraver/paramedir data extraction and, when required, Dimemas simulation:



6.1.1 Measured execution

The original Paraver trace represents the measured application execution. BasicAnalysis uses `paramedir` together with a set of Paraver configuration files to extract the raw information required by the different metric models.

The extracted information depends on the programming model and may include execution times, useful computation, communication behavior, runtime activity, and accelerator activity.

6.1.2 Idealized execution

Some communication-efficiency decompositions require a comparison with an idealized execution.

When these metrics are required, BasicAnalysis converts the relevant trace information for Dimemas and simulates an idealized execution. `paramedir` is then used again to extract the corresponding simulated information.

Measured and simulated data are subsequently combined to compute the efficiency factors that depend on this comparison.

6.1.3 Dimemas ideal configuration

The Dimemas simulation used by BasicAnalysis is controlled by an ideal configuration that represents the assumptions of the idealized communication model.

At the MPI communication level, the ideal configuration assumes zero network latency and infinite network bandwidth. This removes the communication cost associated with message transfer through the interconnection network and provides the idealized reference execution required by the communication- efficiency decomposition.

The purpose of this configuration is not to reproduce the measured machine exactly, but to provide the reference execution required by the corresponding efficiency decomposition.

When simulation is disabled with `--skip-simulation`, metrics requiring idealized execution information cannot be computed and are reported as unavailable.

6.2 Simple Metrics

BasicAnalysis uses the simple metric model for applications whose performance can be analyzed using a single parallel execution domain. This includes, for example, MPI-only and OpenMP-only applications.

The model follows the hierarchical decomposition:

$$Global_Eff = Par_Eff \cdot CompScale$$

and:

$$Par_Eff = LB_Eff \cdot Comm_Eff$$

For MPI applications, Communication Efficiency can additionally be decomposed as:

$$Comm_Eff = Ser_Eff \cdot Transfer_Eff$$

Serialization Efficiency and Transfer Efficiency require the MPI ideal simulation described above and are therefore available only when the required simulation can be performed.

6.2.1 Notation

The following notation is used for the simple metric definitions.

Symbol	Description
T	Application execution time represented by the analyzed trace. BasicAnalysis obtains this value using the <code>runtime_app.cfg</code> Paraver configuration.
P	Number of parallel execution units considered by the simple metric model.
i	Index of a parallel execution unit, $i = 1, \dots, P$.
$Useful_i$	Useful computation duration of parallel execution unit i .
T_{ideal}	Application execution time of the idealized trace generated by Dimemas.
$\max_{ideal}(Useful)$	Maximum useful computation duration obtained from the idealized Dimemas trace.
$Instructions$	Number of processor instructions collected during useful computation.
$Cycles$	Number of processor cycles collected during useful computation.
IPC	Instructions per cycle during useful computation.
$Frequency$	Average CPU frequency during useful computation.
(0)	Reference execution.
(n)	Execution being evaluated.

For the generic simple model, P denotes the parallel execution units used by the corresponding analysis. Its concrete meaning therefore depends on the programming model.

Hardware-counter metrics are available only when the corresponding information is present in the trace.

6.2.2 Parallel Efficiency

Parallel Efficiency measures the fraction of the application execution that corresponds, on average, to useful computation across the parallel execution units.

BasicAnalysis computes:

$$Par_Eff = \frac{\sum_{i=1}^P Useful_i}{P \cdot T}$$

Parallel Efficiency is also computed through the multiplicative decomposition:

$$Par_Eff = LB_Eff \cdot Comm_Eff$$

Load Balance

Load Balance measures how evenly useful computation is distributed across the parallel execution units.

BasicAnalysis computes:

$$LB_Eff = \frac{\frac{1}{P} \sum_{i=1}^P Useful_i}{\max(Useful_1, \dots, Useful_P)}$$

A value close to 100% indicates that useful computation is similarly distributed among the parallel execution units. Lower values indicate increasing workload imbalance.

Communication Efficiency

Communication Efficiency measures the part of the application execution time that remains after accounting for the longest useful-computation path.

BasicAnalysis computes:

$$Comm_Eff = \frac{\max(Useful_1, \dots, Useful_P)}{T}$$

A value close to 100% indicates that little execution time is lost outside the longest useful-computation path. Lower values indicate increasing overhead associated with communication, synchronization, or other parallel-runtime effects.

For MPI applications, BasicAnalysis can further decompose Communication Efficiency into Serialization Efficiency and Transfer Efficiency.

Serialization Efficiency

Serialization Efficiency characterizes the communication-related loss that remains after the MPI communication environment has been idealized.

BasicAnalysis computes:

$$Ser_Eff = \frac{\max_{ideal}(Useful)}{T_{ideal}}$$

where $\max_{ideal}(Useful)$ is the maximum useful-computation duration obtained from the idealized execution and T_{ideal} is its application execution time.

Lower Serialization Efficiency indicates increasing losses associated with dependencies and serialization among MPI processes.

If the resulting Serialization Efficiency exceeds 100%, the measured and simulated executions do not provide a valid conventional efficiency decomposition. BasicAnalysis reports **Warning!** instead of the numerical value.

Transfer Efficiency

Transfer Efficiency characterizes the additional communication degradation present in the measured execution relative to the idealized communication scenario.

Under the standard decomposition, BasicAnalysis derives Transfer Efficiency from Communication Efficiency and Serialization Efficiency:

$$Transfer_Eff = \frac{Comm_Eff}{Ser_Eff}$$

therefore preserving the multiplicative relationship:

$$Comm_Eff = Ser_Eff \cdot Transfer_Eff$$

If Serialization Efficiency is reported as **Warning!**, BasicAnalysis cannot use this decomposition and instead computes Transfer Efficiency directly from the measured and idealized execution times:

$$Transfer_Eff = \frac{T_{ideal}}{T}$$

where T is the measured application execution time.

A lower Transfer Efficiency indicates increasing communication-transfer cost relative to the idealized execution.

If the resulting Transfer Efficiency exceeds 100%, BasicAnalysis reports **Warning!** instead of the numerical value.

Serialization Efficiency and Transfer Efficiency are available when the required MPI ideal simulation can be performed.

The consistency and validity of simulation-derived communication submetrics are discussed in [Simulation Validity](#).

Simulation Validity

Serialization Efficiency and Transfer Efficiency depend on the consistency of the measured and idealized executions.

BasicAnalysis checks the resulting simulation-derived values. When MPI Serialization Efficiency exceeds 100%, BasicAnalysis reports **Warning!** instead of interpreting the value as a conventional efficiency. In this case, MPI Transfer Efficiency is also reported as **Warning!** because the multiplicative communication-efficiency decomposition is not considered valid.

When MPI Serialization Efficiency is valid but the resulting MPI Transfer Efficiency exceeds 100%, MPI Transfer Efficiency is reported as **Warning!**.

The communication submetrics are reported as **Non-Avail** when the required simulation is unavailable, explicitly skipped, or unsupported for the detected programming model.

Computation Scalability

Computation Scalability characterizes how the useful computation evolves relative to a reference execution as the execution configuration changes.

BasicAnalysis uses the total useful-computation duration, $Useful_{tot}$, and distinguishes between strong- and weak-scaling experiments.

For strong scaling:

$$CompScale = \frac{Useful_{tot,ref}}{Useful_{tot}}$$

For weak scaling:

$$CompScale = \frac{Useful_{tot,ref}}{Useful_{tot}} \cdot \frac{P}{P_{ref}}$$

where P is the number of parallel execution units and the **ref** subscript denotes the reference execution.

In strong scaling, the problem size is assumed to remain constant, so changes in total useful-computation time characterize the scalability of the computation.

In weak scaling, the amount of work is expected to grow with the number of parallel resources. The factor P/P_{ref} compensates for this expected increase, allowing Computation Scalability to characterize changes in the computational behavior relative to the weak-scaling assumption.

A value close to 100% indicates that the useful computation scales as expected under the selected scaling model. Lower values indicate increasing computation-related scalability loss.

Computation Scalability is a relative metric and therefore requires multiple execution configurations. The reference execution establishes the baseline and has a Computation Scalability of 100%.

6.2.3 Computation Scalability submetrics

When the required hardware counters are available, BasicAnalysis provides additional factors that decompose changes in Computation Scalability into contributions associated with instruction throughput, the amount of executed instructions, and processor frequency.

The three factors form the computation-scalability decomposition used by BasicAnalysis:

$$CompScale = IPCScale \cdot InstructionScale \cdot FrequencyScale$$

The factors should be interpreted together when investigating a degradation in Computation Scalability.

IPC Scalability

IPC Scalability characterizes changes in instruction throughput during useful computation relative to the reference execution.

BasicAnalysis computes:

$$IPCScale = \frac{IPC}{IPC_{ref}}$$

The IPC used by BasicAnalysis is obtained from the hardware counters collected during useful computation:

$$IPC = \frac{Instructions}{Cycles}$$

where *Instructions* is the number of instructions executed during useful computation and *Cycles* is the corresponding number of processor cycles.

Values below 100% indicate that fewer instructions are completed per cycle than in the reference execution. Values above 100% indicate an improvement in instruction throughput relative to the reference.

Instruction Scalability

Instruction Scalability characterizes changes in the amount of processor instructions required to perform the useful computation.

For strong scaling:

$$InstructionScale = \frac{Instructions_{ref}}{Instructions}$$

For weak scaling:

$$InstructionScale = \frac{Instructions_{ref}}{Instructions} \cdot \frac{P_{ins}}{P_{ins,ref}}$$

where P_{ins} is the number of execution units contributing valid instruction-counter measurements.

The weak-scaling correction accounts for the expected increase in the amount of work as the number of parallel resources grows.

Values below 100% indicate that the execution requires more instructions than expected relative to the reference configuration. This may reflect changes in the executed code path, additional computational work, or other effects that increase the instruction count.

Frequency Scalability

Frequency Scalability characterizes changes in average processor frequency during useful computation relative to the reference execution.

BasicAnalysis computes:

$$FrequencyScale = \frac{Frequency}{Frequency_{ref}}$$

The average processor frequency used by BasicAnalysis is derived from the number of processor cycles measured during useful computation and the corresponding useful-computation time:

$$Frequency = \frac{Cycles}{Useful\ Time}$$

BasicAnalysis reports the resulting average frequency in GHz after applying the corresponding unit conversion.

Values below 100% indicate that the processor operates at a lower average frequency during useful computation than in the reference execution. Values above 100% indicate a higher average frequency relative to the reference.

Frequency changes may result from hardware power-management mechanisms, thermal constraints, processor utilization, or other system-level effects. The metric identifies the contribution of frequency variation to Computation Scalability but does not by itself determine the cause of that variation.

6.3 Hybrid MPI+X Metrics for CPU-Based Applications

For CPU-based hybrid applications, BasicAnalysis extends the application-level performance model by separating the contribution of MPI from that of the inner parallel runtime.

The hybrid application-level model follows the hierarchy:

$$Global_Eff = Hybrid_Par_Eff \cdot CompScale$$

Hybrid Parallel Efficiency is decomposed into the contributions of the outer MPI runtime and the inner parallel runtime:

$$Hybrid_Par_Eff = MPI_Par_Eff \cdot Inner_Par_Eff$$

MPI metrics characterize the outer MPI level, while derived runtime contributions quantify the multiplicative contribution of the inner parallel level to Hybrid Parallel Efficiency.

Computation Scalability and its IPC, Instruction, and Frequency scalability submetrics follow the same formulation used by the simple metric model. For CPU-based hybrid applications, the number of parallel units is the sum of the parallel units of both parallel runtime levels.

This section describes the Computation Scalability, MPI runtime metrics, the Dimemas simulations used to derive communication submetrics, and the multiplicative decomposition used to obtain the contribution of the inner parallel level.

6.3.1 Computation Scalability

For CPU-based MPI+X applications, **Computation Scalability** and its associated IPC, Instruction, and Frequency scalability submetrics are computed using the same formulation defined for the simple metric model.

The difference lies in the parallel units used by the hybrid model. For a CPU-based hybrid execution, the total number of parallel units combines the parallel units associated with both parallel runtime levels.

BasicAnalysis uses the total useful-computation duration, $Useful_{tot}$.

For strong scaling:

$$CompScale = \frac{Useful_{tot,ref}}{Useful_{tot}}$$

For weak scaling:

$$CompScale = \frac{Useful_{tot,ref}}{Useful_{tot}} \cdot \frac{P}{P_{ref}}$$

where P represents the total parallel units of the hybrid execution and the `ref` subscript denotes the reference execution.

The corresponding IPC Scalability, Instruction Scalability, and Frequency Scalability factors are computed using the same definitions described in [Computation Scalability submetrics](#) for the simple metric model.

These factors preserve the computation-scalability decomposition:

$$CompScale = IPCScale \cdot InstructionScale \cdot FrequencyScale$$

6.3.2 MPI runtime metrics

At the MPI level, BasicAnalysis evaluates the execution from the perspective of the MPI ranks using the time spent outside MPI calls. This allows the MPI contribution to Hybrid Parallel Efficiency to be characterized independently of the inner parallel runtime.

The MPI component follows the hierarchy:

$$MPI_Par_Eff = MPI_LB_Eff \cdot MPI_Comm_Eff$$

When the required Dimemas ideal simulation is available, MPI Communication Efficiency can additionally be decomposed as:

$$MPI_Comm_Eff = MPI_Ser_Eff \cdot MPI_Transfer_Eff$$

The MPI component of the hybrid model introduces the following notation.

Symbol	Description
T	Application execution time represented by the analyzed trace.
P	Number of MPI ranks.
i	MPI-rank index, $i = 1, \dots, P$.
$OutsideMPI_i$	Time spent outside MPI calls by MPI rank i .
t_i	Number of host execution units associated with MPI rank i for the OutsideMPI measurement.
T_{ideal}	Application execution time of the MPI-ideal trace generated by Dimemas.
$\max_{ideal}(OutsideMPI)$	Maximum time outside MPI calls obtained from the MPI-ideal simulated trace.

The $OutsideMPI_i$ quantity is used instead of the useful-computation time employed by the simple metric model because the purpose of this decomposition is to isolate the contribution of the MPI runtime within a hybrid application.

BasicAnalysis extracts one representative OutsideMPI duration for each MPI rank. When MPI ranks contain different numbers of host execution units, the OutsideMPI contribution is weighted by the number of host execution units associated with each rank when computing the average contribution used by MPI Parallel Efficiency and MPI Load Balance.

6.3.3 MPI Parallel Efficiency

MPI Parallel Efficiency measures the efficiency associated with the MPI component of the hybrid execution.

For the general case, where MPI ranks may have different numbers of host execution units, BasicAnalysis computes:

$$MPI_Par_Eff = \frac{\sum_{i=1}^P OutsideMPI_i \cdot t_i}{T \cdot \sum_{i=1}^P t_i}$$

The weighting by t_i accounts for configurations in which different MPI ranks contain different numbers of host execution units.

When all MPI ranks have the same number of host execution units, the expression reduces to:

$$MPI_Par_Eff = \frac{\sum_{i=1}^P OutsideMPI_i}{P \cdot T}$$

MPI Parallel Efficiency is decomposed as:

$$MPI_Par_Eff = MPI_LB_Eff \cdot MPI_Comm_Eff$$

MPI Load Balance

MPI Load Balance measures how evenly the time outside MPI calls is distributed across the MPI ranks.

For the general case, BasicAnalysis computes:

$$MPI_LB_Eff = \frac{\frac{\sum_{i=1}^P OutsideMPI_i \cdot t_i}{\sum_{i=1}^P t_i}}{\max(OutsideMPI_1, \dots, OutsideMPI_P)}$$

The numerator therefore represents the OutsideMPI duration averaged over the host execution units associated with the MPI ranks.

When all MPI ranks have the same number of host execution units, this reduces to:

$$MPI_LB_Eff = \frac{\frac{1}{P} \sum_{i=1}^P OutsideMPI_i}{\max(OutsideMPI_1, \dots, OutsideMPI_P)}$$

A value close to 100% indicates that the time outside MPI is similarly distributed among MPI ranks. Lower values indicate increasing imbalance between ranks from the MPI-runtime perspective.

MPI Communication Efficiency

MPI Communication Efficiency characterizes the communication and synchronization overhead observed from the MPI perspective by comparing the maximum time spent outside MPI among the ranks with the application execution time.

BasicAnalysis computes:

$$MPI_Comm_Eff = \frac{\max(OutsideMPI_1, \dots, OutsideMPI_P)}{T}$$

A value close to 100% indicates that the rank with the largest OutsideMPI duration spends most of the application execution outside MPI. Lower values indicate increasing time associated with MPI communication or synchronization on the critical MPI path.

Together with MPI Load Balance, this preserves the multiplicative relationship:

$$MPI_Par_Eff = MPI_LB_Eff \cdot MPI_Comm_Eff$$

When a supported Dimemas ideal simulation is available, MPI Communication Efficiency can be further decomposed into MPI Serialization Efficiency and MPI Transfer Efficiency.

MPI Serialization Efficiency

MPI Serialization Efficiency characterizes the communication-related loss that remains from the MPI perspective after the MPI communication environment has been idealized.

BasicAnalysis computes:

$$MPI_Ser_Eff = \frac{\max_{ideal}(OutsideMPI)}{T_{ideal}}$$

where $\max_{ideal}(OutsideMPI)$ is the maximum time spent outside MPI among the MPI ranks in the idealized execution and T_{ideal} is the application execution time of that execution.

A lower MPI Serialization Efficiency indicates increasing communication loss associated with dependencies and temporal serialization among MPI ranks that remain after idealizing the MPI communication environment.

MPI Transfer Efficiency

MPI Transfer Efficiency characterizes the additional MPI communication degradation associated with the non-ideal communication environment of the measured execution.

BasicAnalysis derives MPI Transfer Efficiency from MPI Communication Efficiency and MPI Serialization Efficiency:

$$MPI_Transfer_Eff = \frac{MPI_Comm_Eff}{MPI_Ser_Eff}$$

therefore preserving the multiplicative relationship:

$$MPI_Comm_Eff = MPI_Ser_Eff \cdot MPI_Transfer_Eff$$

A lower MPI Transfer Efficiency indicates an increasing contribution from the communication-transfer costs represented by the difference between the measured and MPI-ideal communication environments.

Unlike the corresponding simple-model expression, the hybrid formulation cannot in general be simplified to:

$$\frac{T_{ideal}}{T}$$

because MPI Communication Efficiency and MPI Serialization Efficiency are defined using the maximum OutsideMPI duration of the measured and idealized executions, respectively:

$$MPI_Transfer_Eff = \frac{\max(OutsideMPI)/T}{\max_{ideal}(OutsideMPI)/T_{ideal}}$$

and $\max(OutsideMPI)$ is not assumed to be equal to $\max_{ideal}(OutsideMPI)$.

MPI Serialization Efficiency and MPI Transfer Efficiency are available when the required MPI-ideal Dimemas simulation can be performed for the corresponding hybrid programming model.

The consistency and validity of simulation-derived communication submetrics are discussed in *Simulation Validity*.

6.3.4 Dimemas simulation

For CPU-based hybrid applications, the MPI-level simulation uses the ideal MPI communication configuration described above while disabling the inner parallel runtime.

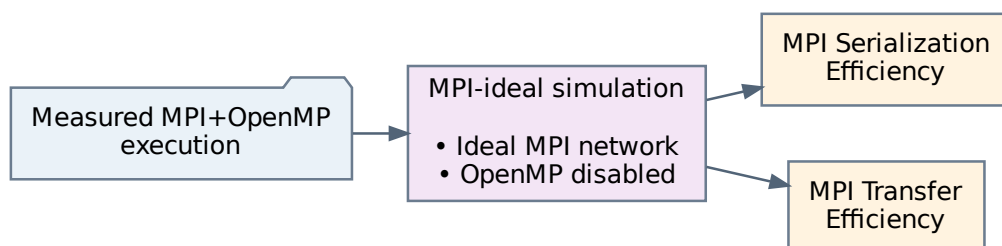
This allows the simulation-derived quantities to be interpreted from the MPI perspective, consistently with the use of OutsideMPI in MPI Serialization Efficiency and MPI Transfer Efficiency.

BasicAnalysis currently applies this standard simulation methodology to MPI+OpenMP applications.

MPI+OpenMP

For the standard MPI-level analysis of an MPI+OpenMP application, BasicAnalysis invokes Dimemas with OpenMP simulation disabled.

Conceptually, the experiment is:



The resulting idealized execution is used to obtain T_{ideal} and $\max_{ideal}(OutsideMPI)$, which are required to compute MPI Serialization Efficiency and MPI Transfer Efficiency.

OpenMP is disabled in this experiment because the objective is to isolate the MPI communication contribution rather than to model the combined MPI+OpenMP execution.

Optional MPI+OpenMP Hybrid Simulation

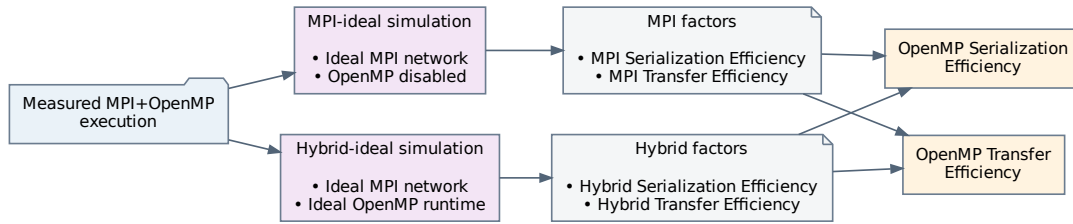
For MPI+OpenMP applications, BasicAnalysis can optionally perform a second ideal simulation when `--hyb-mpiomp` is enabled.

This experiment is different from the standard MPI-level simulation described above. The standard simulation disables OpenMP in order to isolate the MPI communication contribution. The optional hybrid simulation instead enables the Dimemas ideal-OpenMP model while retaining the idealized MPI communication environment.

BasicAnalysis therefore generates two different idealized executions:

- an **MPI-ideal execution**, used to compute MPI Serialization Efficiency and MPI Transfer Efficiency; and
- a **hybrid-ideal execution**, used to characterize the communication decomposition when both the MPI communication environment and the OpenMP execution are idealized according to the corresponding Dimemas models.

Conceptually:



From the hybrid-ideal trace, BasicAnalysis extracts the maximum useful- computation duration and the application execution time. These quantities are used to compute an intermediate hybrid Serialization Efficiency:

$$Hybrid_Ser_Eff = \frac{\max_{ideal,hybrid}(Useful)}{T_{ideal,hybrid}}$$

where $\max_{ideal,hybrid}(Useful)$ is the maximum useful-computation duration obtained from the hybrid-ideal trace and $T_{ideal,hybrid}$ is its execution time.

BasicAnalysis then derives the corresponding hybrid Transfer Efficiency from the application-level Communication Efficiency of the measured hybrid execution:

$$Hybrid_Transfer_Eff = \frac{Hybrid_Comm_Eff}{Hybrid_Ser_Eff}$$

These hybrid quantities are intermediate values used to derive the OpenMP contribution to the communication decomposition. They should not be confused with the MPI Serialization and Transfer Efficiencies obtained from the standard MPI-level simulation.

Their use in deriving the OpenMP runtime contribution is described in [OpenMP communication submetrics](#).

Advanced OpenMP Dimemas Simulation Options

BasicAnalysis provides advanced options that allow users to modify how OpenMP is treated in the Dimemas simulation.

These options alter the simulated experiment. Consequently, metrics derived from the resulting simulated trace do not necessarily have the same interpretation as the standard BasicAnalysis Serialization Efficiency and Transfer Efficiency metrics described above.

`-somp, --simulation_omp`

Enables the simulation of OpenMP events for MPI+OpenMP applications. Without this option, OpenMP simulation is disabled in the standard MPI-level experiment.

`--ideal-omp`

Uses the Dimemas ideal-OpenMP model during an MPI+OpenMP simulation. OpenMP runtime-event duration is ignored, while the remaining duration is associated with implicit synchronization.

This option modifies how OpenMP is modeled and is meaningful when OpenMP simulation is enabled.

These options are intended for users who understand the corresponding Dimemas simulation models and want to perform customized simulation experiments.

The standard BasicAnalysis methodology uses the simulation configuration selected internally for each supported programming model. Enabling these advanced options changes the assumptions of the idealized execution and may therefore change the meaning and interpretation of the simulation-derived metrics.

6.3.5 Derived Inner-Runtime Contributions

For CPU-based hierarchical hybrid applications, BasicAnalysis derives the contribution of the inner parallel level from the application-level and MPI runtime metrics.

The model assumes the multiplicative decomposition:

$$Hybrid_Par_Eff = MPI_Par_Eff \cdot Inner_Par_Eff$$

Therefore:

$$Inner_Par_Eff = \frac{Hybrid_Par_Eff}{MPI_Par_Eff}$$

Similarly:

$$Inner_LB_Eff = \frac{Hybrid_LB_Eff}{MPI_LB_Eff}$$

and:

$$Inner_Comm_Eff = \frac{Hybrid_Comm_Eff}{MPI_Comm_Eff}$$

These quantities are derived multiplicative contributions of the inner parallel level and do not require the inner runtime to be isolated directly from the trace.

MPI+OpenMP Runtime Contribution

For the currently supported MPI+OpenMP case, the generic inner-runtime contributions defined above correspond to the OpenMP runtime contribution:

$$Inner_Par_Eff \equiv OMP_Par_Eff$$

$$Inner_LB_Eff \equiv OMP_LB_Eff$$

$$Inner_Comm_Eff \equiv OMP_Comm_Eff$$

Therefore, the application-level decomposition can be expressed as:

$$Hybrid_Par_Eff = MPI_Par_Eff \cdot OMP_Par_Eff$$

$$Hybrid_LB_Eff = MPI_LB_Eff \cdot OMP_LB_Eff$$

$$Hybrid_Comm_Eff = MPI_Comm_Eff \cdot OMP_Comm_Eff$$

The application-level metrics in these expressions are those measured for the complete MPI+OpenMP execution, while the MPI metrics represent the MPI contribution isolated through the OutsideMPI-based decomposition described earlier.

The resulting OpenMP factors therefore represent the multiplicative contribution required to relate the MPI-level metrics to the corresponding application-level metrics.

OpenMP communication submetrics

When the optional MPI+OpenMP hybrid simulation is enabled with `--hyb-mpiomp`, BasicAnalysis can also derive the OpenMP contribution to Serialization Efficiency and Transfer Efficiency.

The OpenMP Serialization Efficiency contribution is derived as:

$$OMP_Ser_Eff = \frac{Hybrid_Ser_Eff}{MPI_Ser_Eff}$$

and the OpenMP Transfer Efficiency contribution as:

$$OMP_Transfer_Eff = \frac{Hybrid_Transfer_Eff}{MPI_Transfer_Eff}$$

Therefore, the multiplicative relationships are:

$$Hybrid_Ser_Eff = MPI_Ser_Eff \cdot OMP_Ser_Eff$$

and:

$$Hybrid_Transfer_Eff = MPI_Transfer_Eff \cdot OMP_Transfer_Eff$$

The OpenMP Serialization and Transfer Efficiencies obtained in this way are **derived runtime contributions**. They are not independent measurements of an OpenMP-only execution.

These metrics are available only for MPI+OpenMP applications when `--hyb-mpiomp` is explicitly requested, the required Dimemas simulations complete successfully, and the corresponding simulated-trace measurements can be obtained.

Interpretation of Derived Runtime Contributions

The runtime factors above are multiplicative contributions relative to the MPI component. They are not standalone efficiencies obtained from an independent OpenMP execution.

Because they are obtained as ratios between application-level and MPI-level metrics, a derived contribution is not mathematically restricted to 100%.

For example:

$$OMP_LB_Eff = \frac{Hybrid_LB_Eff}{MPI_LB_Eff}$$

can exceed 100% when the application-level Load Balance is greater than the MPI Load Balance.

A value above 100% must therefore not be interpreted as an execution efficiency above the theoretical maximum. Instead, it indicates that the application-level metric is greater than the corresponding MPI-level metric, resulting in a multiplicative inner-runtime contribution above 100%.

6.4 Hybrid MPI+GPU Metrics

For MPI+GPU applications, BasicAnalysis extends the hybrid performance model to account for both the Host and accelerator components of the execution.

The analysis provides two complementary views:

- the **Parallel Runtime Model**, which separates the contributions of the MPI and CUDA/HIP runtimes to the hybrid application-level efficiency; and
- the **Host/Device Execution Domains**, which characterize where accelerator-related inefficiency manifests in the physical execution domains.

The construction of the GPU-related application-level and execution-domain quantities relies on the correct association of GPU stream activity with the corresponding physical devices.

6.4.1 Stream-to-device activity aggregation

GPU traces may contain several execution streams associated with the same physical accelerator. BasicAnalysis therefore does not treat individual GPU streams as independent physical execution resources when constructing the MPI+GPU metrics.

Instead, GPU stream activity is first mapped to the corresponding physical devices using the device information available in the Paraver trace metadata.

For each physical device d , BasicAnalysis collects the useful- computation intervals from all streams mapped to that device and computes their temporal union. Overlapping intervals are therefore counted only once.

The resulting useful-computation duration of device d is:

$$Useful_d = \left| \bigcup_{s \in Streams(d)} UsefulIntervals_s \right|$$

where $Streams(d)$ denotes the set of GPU streams mapped to physical device d .

The aggregate useful device computation is then:

$$UsefulDevice = \sum_{d=1}^D Useful_d$$

where D is the number of physical devices.

BasicAnalysis also obtains:

$$UsefulDevice_{max} = \max_d(Useful_d)$$

This stream-to-device mapping and interval flattening prevents concurrent activity occurring on different streams of the same physical device from being double-counted.

Device memory-transfer activity

BasicAnalysis applies the same physical-device mapping principle to memory-transfer activity.

For each physical device, memory-transfer intervals from all associated GPU streams are collected and flattened. BasicAnalysis then removes any portion of the resulting transfer activity that overlaps useful device computation.

Only memory-transfer activity occurring outside useful computation therefore contributes additional device active time.

For device d :

$$UsefulMemTransfer_d = Useful_d + NonOverlappingTransfer_d$$

where $NonOverlappingTransfer_d$ is the duration of memory-transfer activity that does not overlap useful computation on that physical device.

BasicAnalysis then obtains:

$$UsefulMemTransfer_{max} = \max_d(UsefulMemTransfer_d)$$

This construction prevents useful computation and memory-transfer activity occurring concurrently on different streams of the same physical device from being double-counted.

6.4.2 MPI+GPU Application-Level Metrics

For MPI+GPU applications, BasicAnalysis constructs hybrid application-level metrics that account for useful computation performed in both the Host and Device execution domains.

Let:

- $UsefulHost$ denote the aggregate useful Host computation;

- *UsefulDevice* denote the aggregate useful Device computation after stream-to-device mapping and interval flattening;
- *H* denote the number of Host execution units;
- *D* denote the number of physical devices; and
- *T* denote the application execution time.

The hybrid number of parallel units is therefore:

$$H + D$$

rather than the number of Host execution units plus the number of GPU streams.

Computation Scalability

For MPI+GPU applications, **Computation Scalability** characterizes how the useful computation of the complete hybrid execution evolves relative to the reference execution.

The application-level useful computation combines Host and Device useful computation:

$$Useful_{app} = Useful_{Host} + Useful_{Device}$$

where *UsefulDevice* is the total useful Device computation obtained after GPU streams have been mapped to their corresponding physical devices and overlapping useful-computation intervals on each device have been flattened as described in [Stream-to-device activity aggregation](#).

GPU streams are therefore not treated as independent parallel execution units when constructing either the Device useful-computation quantity or the application-level parallel units.

For strong scaling, BasicAnalysis computes:

$$CompScale = \frac{Useful_{app,ref}}{Useful_{app}}$$

For weak scaling, the application-level parallel units is:

$$P_{app} = H + D$$

where *H* is the number of Host execution units and *D* is the number of physical devices.

BasicAnalysis therefore computes:

$$CompScale = \frac{Useful_{app,ref}}{Useful_{app}} \cdot \frac{H + D}{H_{ref} + D_{ref}}$$

This normalization uses the same physical parallel units as the MPI+GPU application-level model. In particular, GPU streams are not counted as independent Device resources; the Device contribution is represented by the number of physical devices.

Within the Parallel Runtime Model, Computation Scalability is reported as a single factor for the complete MPI+GPU execution. Unlike the CPU-based hybrid model, BasicAnalysis does not decompose it into IPC, Instruction, and Frequency scalability submetrics.

The Host/Device Execution Domains provide a complementary execution-domain view of Computation Scalability. In that view, Host Computation Scalability and Device Computation Scalability are computed independently using the parallel units associated with each physical execution domain: Host execution units for the Host domain and physical devices for the Device domain.

Hybrid Parallel Efficiency

BasicAnalysis computes:

$$Hybrid_Par_Eff = \frac{Useful_{Host} + Useful_{Device}}{(H + D)T}$$

Hybrid Load Balance

Let:

$$Useful_{max} = \max(UsefulHost_{max}, UsefulDevice_{max})$$

where $UsefulHost_{max}$ is the maximum useful-computation duration among the Host execution units and $UsefulDevice_{max}$ is the maximum flattened useful-computation duration among the physical devices.

BasicAnalysis computes:

$$Hybrid_LB_Eff = \frac{\frac{UsefulHost + UsefulDevice}{H + D}}{Useful_{max}}$$

Hybrid Communication Efficiency

BasicAnalysis computes:

$$Hybrid_Comm_Eff = \frac{Useful_{max}}{T}$$

The hybrid application-level metrics preserve the multiplicative decomposition:

$$Hybrid_Par_Eff = Hybrid_LB_Eff \cdot Hybrid_Comm_Eff$$

These quantities characterize the complete MPI+GPU execution and provide the application-level reference used by the Parallel Runtime Model.

6.4.3 MPI+GPU Parallel Runtime Model

The Parallel Runtime Model separates the contribution of the outer MPI runtime from that of the accelerator runtime.

The MPI contribution is characterized using the MPI runtime metrics described for the hybrid model. For MPI+GPU applications, however, `OutsideMPI` is treated strictly as an MPI-rank quantity. BasicAnalysis uses one representative `OutsideMPI` value per MPI rank, independently of the number of GPU streams associated with that rank. Consequently, MPI Parallel Efficiency, MPI Load Balance, and MPI Communication Efficiency are computed over the MPI ranks; GPU streams do not contribute to the number of parallel units of these MPI-level metrics.

The MPI-level decomposition remains:

$$MPI_Par_Eff = MPI_LB_Eff \cdot MPI_Comm_Eff$$

The accelerator-runtime contribution is then derived as the multiplicative contribution required to relate the MPI-level metrics to the corresponding hybrid application-level metrics.

For Parallel Efficiency:

$$Hybrid_Par_Eff = MPI_Par_Eff \cdot GPU_Par_Eff$$

therefore:

$$GPU_Par_Eff = \frac{Hybrid_Par_Eff}{MPI_Par_Eff}$$

For Load Balance:

$$Hybrid_LB_Eff = MPI_LB_Eff \cdot GPU_LB_Eff$$

therefore:

$$GPU_LB_Eff = \frac{Hybrid_LB_Eff}{MPI_LB_Eff}$$

For Communication Efficiency:

$$Hybrid_Comm_Eff = MPI_Comm_Eff \cdot GPU_Comm_Eff$$

therefore:

$$GPU_Comm_Eff = \frac{Hybrid_Comm_Eff}{MPI_Comm_Eff}$$

Here GPU represents the CUDA or HIP runtime according to the programming model detected in the trace.

These accelerator-runtime factors are **derived multiplicative contributions**, analogous to the derived inner-runtime contributions of the CPU-based hybrid model. They are not independent measurements of a CUDA-only or HIP-only execution and are therefore not mathematically restricted to 100%.

The runtime contributions must not be confused with the metrics of the Device Execution Domain. The Parallel Runtime Model attributes efficiency losses to the MPI and CUDA/HIP runtime contributions, whereas the Host/Device analysis characterizes where accelerator-related inefficiency manifests in the physical execution domains.

6.4.4 Host and Device Execution Domains

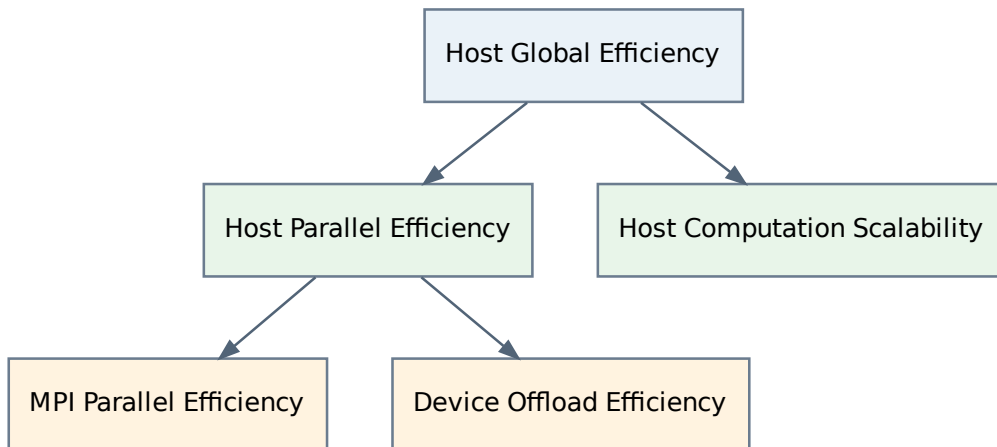
For MPI+CUDA and MPI+HIP applications, BasicAnalysis provides an additional Host/Device decomposition.

This view is complementary to the Parallel Runtime Model. The Parallel Runtime Model attributes efficiency losses to the MPI and CUDA/HIP runtime contributions, while the execution-domain analysis identifies whether accelerator-related inefficiency manifests in the Host or Device execution domain.

Host Execution Domain

The Host domain characterizes useful Host computation and the Host-side interaction with the accelerator.

Its hierarchy is:



The Host Parallel Efficiency hierarchy is:

$$Host_Par_Eff = MPI_Par_Eff \cdot DeviceOffload_Eff$$

with:

$$MPI_Par_Eff = MPI_LB_Eff \cdot MPI_Comm_Eff$$

and, when the required MPI-ideal simulation is available:

$$MPI_Comm_Eff = MPI_Ser_Eff \cdot MPI_Transfer_Eff$$

Device Offload Efficiency

Device Offload Efficiency characterizes the fraction of the time outside MPI that corresponds to useful Host computation.

Let *UsefulHost* denote the accumulated useful Host-computation time and *OutsideMPI* the accumulated OutsideMPI duration across the MPI ranks.

BasicAnalysis computes:

$$DeviceOffload_Eff = \frac{UsefulHost}{OutsideMPI}$$

A low Device Offload Efficiency indicates that a significant fraction of the Host execution outside MPI does not correspond to useful Host computation.

This time can include accelerator-runtime activity such as kernel launches, synchronization, memory-management operations, or other Host-side activity associated with supplying and coordinating work on the accelerator.

Device Offload Efficiency does not measure the efficiency of GPU kernel execution itself. Device-side behavior is characterized separately by the Device Execution Domain.

Host Parallel Efficiency

Host Parallel Efficiency measures the fraction of the available Host execution capacity corresponding to useful Host computation.

BasicAnalysis decomposes it as:

$$Host_Par_Eff = MPI_Par_Eff \cdot DeviceOffload_Eff$$

For a homogeneous Host configuration, this relationship can be expressed as:

$$MPI_Par_Eff = \frac{OutsideMPI}{HT}$$

and:

$$DeviceOffload_Eff = \frac{UsefulHost}{OutsideMPI}$$

therefore:

$$Host_Par_Eff = \frac{UsefulHost}{HT}$$

where *H* is the number of Host execution units.

Host Computation Scalability

Host Computation Scalability measures how the aggregate amount of useful Host computation evolves across execution configurations.

For strong scaling, using configuration 0 as the reference:

$$Host_CompScale_n = \frac{UsefulHost_0}{UsefulHost_n}$$

with:

$$Host_CompScale_0 = 1$$

Under ideal strong scaling, the total Host computational work remains approximately constant and Host Computation Scalability therefore remains close to 100%.

For weak scaling:

$$Host_CompScale_n = \frac{UsefulHost_0}{UsefulHost_n} \cdot \frac{H_n}{H_0}$$

where H_n/H_0 is the ratio of Host execution units between configuration n and the reference execution.

The number of Host execution units is obtained from the Host rank/threads identified in the trace.

Under ideal weak scaling, aggregate useful Host computation grows proportionally with the Host execution resources, resulting in a value close to 100%.

Host Computation Scalability Submetrics

When the required Host-side hardware-counter information is available, BasicAnalysis further decomposes Host Computation Scalability into IPC, Instruction, and Frequency scalability factors:

$$Host_CompScale = Host_IPCScale \cdot Host_InstructionScale \cdot Host_FrequencyScale$$

These factors apply the computation-scalability decomposition to the hardware-counter information associated with useful Host computation.

For weak scaling, Host Instruction Scalability is normalized using the number of Host execution units that provide valid instruction-counter measurements. Under normal trace conditions, this parallel-unit count corresponds to the Host execution units contributing useful computation and the multiplicative relationship is preserved.

If instruction-counter measurements are missing for some Host execution units, the reported child factors may not reproduce Host Computation Scalability exactly. Such a discrepancy indicates incomplete hardware-counter information rather than a different definition of the scalability model.

Host IPC Scalability

Host IPC Scalability characterizes changes in Host instruction throughput during useful Host computation relative to the reference execution.

BasicAnalysis computes:

$$Host_IPCScale = \frac{HostIPC}{HostIPC_{ref}}$$

where:

$$HostIPC = \frac{UsefulInstructions}{UsefulCycles}$$

Values below 100% indicate a reduction in Host instruction throughput relative to the reference execution.

Host Instruction Scalability

Host Instruction Scalability characterizes changes in the amount of processor instructions associated with useful Host computation.

For strong scaling:

$$Host_InstructionScale = \frac{UsefulInstructions_{ref}}{UsefulInstructions}$$

For weak scaling:

$$Host_InstructionScale = \frac{UsefulInstructions_{ref}}{UsefulInstructions} \cdot \frac{P_{ins}}{P_{ins,ref}}$$

where P_{ins} is the number of Host execution units contributing valid instruction-counter measurements.

The weak-scaling correction therefore uses the parallel units of Host execution units contributing valid instruction-counter measurements. Under normal trace conditions, this number corresponds to the number of Host execution units used by Host Computation Scalability.

Host Frequency Scalability

Host Frequency Scalability characterizes changes in the effective processor frequency during useful Host computation relative to the reference execution.

BasicAnalysis computes:

$$Host_FrequencyScale = \frac{HostFrequency}{HostFrequency_{ref}}$$

The effective Host frequency is derived from the Host-side cycle and useful-computation measurements used by the Host model.

Values below 100% indicate that useful Host computation executes at a lower effective processor frequency than in the reference configuration.

The effective Host frequency used for Host Frequency Scalability should not be confused with the Average Frequency reported by BasicAnalysis as a general performance indicator.

Host Global Efficiency

Host Global Efficiency combines Host Parallel Efficiency and Host Computation Scalability:

$$Host_Global_Eff = Host_Par_Eff \cdot Host_CompScale$$

Therefore:

$$Host_Global_Eff = MPI_Par_Eff \cdot DeviceOffload_Eff \cdot Host_CompScale$$

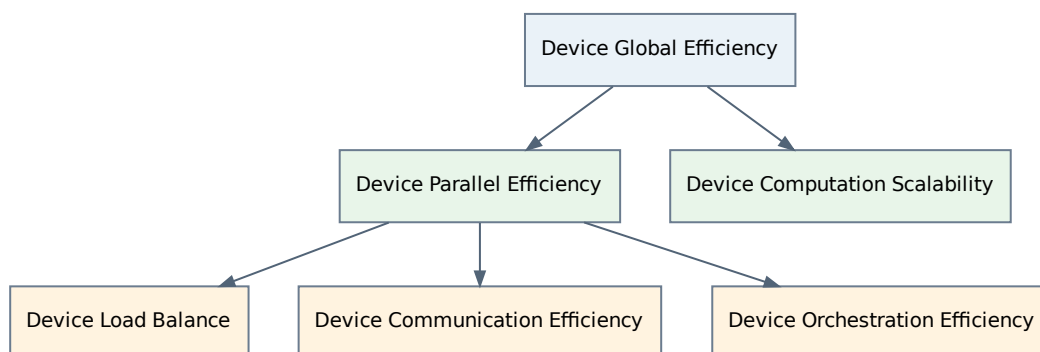
For a single-trace analysis, where Host Computation Scalability cannot be evaluated relative to another execution configuration, BasicAnalysis reports:

$$Host_Global_Eff = Host_Par_Eff$$

Device Execution Domain

The Device domain evaluates useful accelerator activity after all GPU streams have been mapped and flattened to their corresponding physical devices.

Its hierarchy is:



Device Parallel Efficiency

Device Parallel Efficiency measures the fraction of the available physical Device execution capacity corresponding to useful device computation.

BasicAnalysis computes:

$$Device_Par_Eff = \frac{UsefulDevice}{DT}$$

where D is the number of physical devices after stream-to-device mapping.

Device Load Balance

Device Load Balance measures how evenly useful computation is distributed among the physical devices:

$$Device_LB_Eff = \frac{UsefulDevice/D}{UsefulDevice_{max}}$$

A value close to 100% indicates that useful computation is similarly distributed among the physical devices. Lower values indicate increasing imbalance in useful device activity.

Device Communication Efficiency

Device Communication Efficiency characterizes the additional active device time associated with memory transfers that do not overlap useful computation.

BasicAnalysis computes:

$$Device_Comm_Eff = \frac{UsefulDevice_{max}}{UsefulMemTransfer_{max}}$$

The numerator and denominator are independently computed maxima across the physical devices and are not required to correspond to the same device.

A low Device Communication Efficiency indicates that non-overlapped memory-transfer activity contributes substantially to the maximum device active time relative to the maximum useful-computation time observed across the devices.

Device Orchestration Efficiency

Device Orchestration Efficiency measures how much of the application execution time is covered by the longest useful-computation plus non-overlapped-memory-transfer device activity.

BasicAnalysis computes:

$$Device_Orch_Eff = \frac{UsefulMemTransfer_{max}}{T}$$

Lower values indicate increasing periods during which the devices are not performing useful computation or non-overlapped memory-transfer activity.

Such periods can reflect limitations in supplying work to the accelerators, Host-side orchestration delays, synchronization, dependencies, or other effects that leave the devices without active work. The metric identifies the presence of these inactive periods but does not by itself determine their cause.

Device Parallel-Efficiency Decomposition

The three Device factors preserve the multiplicative decomposition:

$$Device_Par_Eff = Device_LB_Eff \cdot Device_Comm_Eff \cdot Device_Orch_Eff$$

Indeed:

$$Device_Par_Eff = \frac{UsefulDevice/D}{UsefulDevice_{max}} \cdot \frac{UsefulDevice_{max}}{UsefulMemTransfer_{max}} \cdot \frac{UsefulMemTransfer_{max}}{T} = \frac{UsefulDevice}{DT}$$

Device Computation Scalability

Device Computation Scalability measures how aggregate useful device computation evolves across execution configurations.

The useful-device quantity used by this metric is the aggregate physical- device activity obtained after stream-to-device mapping and interval flattening:

$$UsefulDevice_n = \sum_{d=1}^{D_n} Useful_{d,n}$$

It is therefore not the sum of the raw useful times of all GPU streams.

For strong scaling:

$$Device_CompScale_n = \frac{UsefulDevice_0}{UsefulDevice_n}$$

with:

$$Device_CompScale_0 = 1$$

Under ideal strong scaling, aggregate useful device computation remains approximately constant.

For weak scaling:

$$Device_CompScale_n = \frac{UsefulDevice_0}{UsefulDevice_n} \cdot \frac{D_n}{D_0}$$

where D_n/D_0 is the ratio of physical devices between configuration n and the reference execution.

Under ideal weak scaling, aggregate useful device computation grows proportionally with the number of physical devices, resulting in a value close to 100%.

Device Global Efficiency

Device Global Efficiency combines Device Parallel Efficiency and Device Computation Scalability:

$$Device_Global_Eff = Device_Par_Eff \cdot Device_CompScale$$

Expanding the Device Parallel Efficiency decomposition:

$$Device_Global_Eff = Device_LB_Eff \cdot Device_Comm_Eff \cdot Device_Orch_Eff \cdot Device_CompScale$$

For a single-trace analysis, where Device Computation Scalability cannot be evaluated relative to another execution configuration, BasicAnalysis reports:

$$Device_Global_Eff = Device_Par_Eff$$

Interpretation of Host and Device Metrics

The Host and Device execution-domain metrics are complementary views of an MPI+GPU execution and should be interpreted together with the application- level and runtime-specific metrics.

The absolute values of Host Global Efficiency and Device Global Efficiency should not be directly compared to determine which execution domain “contributes more” to application performance. The two metrics characterize different parallel resources and different aspects of the heterogeneous execution.

The application-level metrics identify the overall performance loss. The Parallel Runtime Model identifies the runtime contribution associated with parallel inefficiency, while the Host and Device execution-domain metrics localize where accelerator-related inefficiency manifests.

Within the Host domain, Device Offload Efficiency helps distinguish useful Host computation from Host-side accelerator interaction, while the Host Computation Scalability factors characterize changes in the computational behavior of useful Host work.

Within the Device domain, Device Load Balance characterizes the distribution of useful computation across physical accelerators, Device Communication Efficiency characterizes additional active time associated with non-overlapped memory transfers, and Device Orchestration Efficiency characterizes periods in which the devices are not supplied with useful computation or non-overlapped transfer activity.

These views should therefore be used as complementary evidence when investigating the source of a performance loss rather than as interchangeable efficiency measurements.

6.4.5 Dimemas simulation

As for MPI+X CPU-based applications, BasicAnalysis uses Dimemas in MPI+GPU applications to obtain the idealized execution required for the MPI communication-efficiency decomposition.

The MPI-level simulation follows the same methodology: the MPI communication environment is idealized using infinite bandwidth and zero latency, while accelerator simulation is disabled. The resulting simulation-derived quantities can therefore be interpreted from the MPI perspective, consistently with the use of `OutsideMPI` in MPI Serialization Efficiency and MPI Transfer Efficiency.

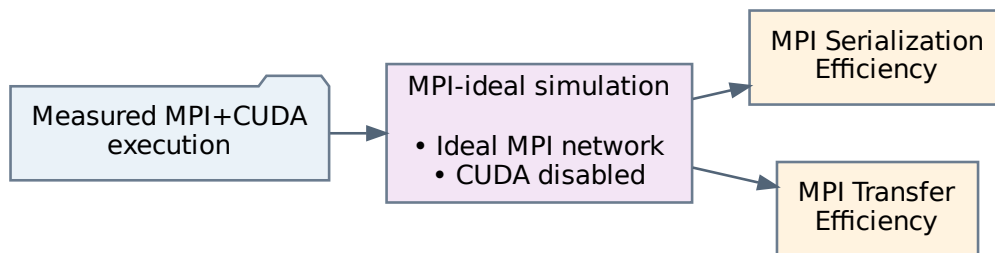
For MPI+GPU applications, these MPI communication submetrics are used wherever the MPI contribution is represented in the analysis. This includes both the MPI contribution of the Parallel Runtime Model and the MPI component of the Host execution-domain hierarchy.

BasicAnalysis currently applies this simulation methodology to MPI+CUDA applications. MPI+HIP is treated differently because the current BasicAnalysis-Dimemas workflow does not provide the corresponding simulation-derived communication submetrics.

MPI+CUDA

For the standard MPI-level analysis of an MPI+CUDA application, BasicAnalysis invokes Dimemas with CUDA simulation disabled.

Conceptually, the experiment is:



The resulting idealized execution provides T_{ideal} and $\max_{ideal}(OutsideMPI)$, which are used to compute MPI Serialization Efficiency and MPI Transfer Efficiency.

Disabling CUDA simulation allows this experiment to isolate the MPI communication contribution without introducing accelerator-simulation effects into the MPI Serialization and Transfer Efficiency submetrics.

CUDA Simulation Option

BasicAnalysis provides the `-scuda`, `--simulation_cuda` option to enable CUDA simulation in Dimemas.

This option changes the simulated experiment from the standard MPI-level simulation used by the BasicAnalysis performance model. In the standard methodology, CUDA simulation is disabled so that the idealized execution isolates the MPI communication contribution.

When CUDA simulation is explicitly enabled, the resulting simulated execution also incorporates the CUDA simulation model. Consequently, performance metrics derived from this simulated execution do not represent the standard BasicAnalysis MPI communication-efficiency decomposition and should not be interpreted as the MPI Serialization Efficiency and MPI Transfer Efficiency defined by the performance model described in this chapter.

This option is therefore intended for customized simulation experiments rather than for obtaining the standard BasicAnalysis performance metrics.

MPI+HIP

MPI Communication Efficiency for MPI+HIP applications is computed directly from the measured execution and therefore does not require Dimemas.

However, MPI Serialization Efficiency and MPI Transfer Efficiency require the MPI-ideal simulation described above. The current BasicAnalysis-Dimemas workflow does not provide the corresponding simulation-derived communication submetrics for HIP executions. These metrics are therefore reported as `Non-Avail` for MPI+HIP.

This limitation affects only the simulation-derived MPI communication submetrics. Metrics computed directly from the measured MPI+HIP trace, including the available Parallel Runtime Model factors and the Host and Device execution-domain metrics, remain available when their required trace information is present.

6.5 OpenMP Runtime-Specific Metrics

BasicAnalysis provides a set of OpenMP runtime-specific efficiency metrics for **OpenMP-only** and **MPI+OpenMP** applications. These metrics are based on the OpenMP efficiency model introduced by TALP and provide an isolated view of the OpenMP execution.

These metrics must not be confused with the derived OpenMP contribution of the Parallel Runtime Model described in *MPI+OpenMP Runtime Contribution*. That contribution quantifies the multiplicative contribution of OpenMP to the application-level efficiency decomposition.

In contrast, the OpenMP Runtime-Specific metrics are computed from OpenMP-specific timing information extracted from the trace. Their purpose is to characterize the OpenMP execution itself and identify which OpenMP mechanisms contribute to its inefficiency.

The analysis separates three main sources of OpenMP inefficiency:

- serial execution outside OpenMP parallel regions;
- useful-work imbalance among threads inside OpenMP parallel regions; and
- OpenMP runtime scheduling and fork/join overhead.

6.5.1 OpenMP Parallel Efficiency

OpenMP Parallel Efficiency characterizes the efficiency of the OpenMP execution after accounting for the main sources of OpenMP-specific thread-capacity loss.

BasicAnalysis decomposes it as:

$$OMP_PE = OMP_Serial \cdot OMP_LB \cdot OMP_Sched$$

The formulation uses four timing components:

Symbol	Description
T_{noOMP}	Thread-capacity time not attributed to OpenMP-specific losses. For OpenMP-only applications, it corresponds to useful computation. For MPI+OpenMP applications, MPI execution time is also included because MPI activity is not considered an OpenMP runtime loss.
T_{serial}^{OMP}	Thread-capacity time lost because OpenMP parallelism is not active.
T_{Imbal}^{OMP}	Thread-capacity time lost because useful work is unevenly distributed among OpenMP threads inside parallel regions.
T_{sched}^{OMP}	Thread-capacity time associated with OpenMP runtime scheduling and fork/join overhead.

The total OpenMP execution capacity represented by the model is therefore:

$$T_{capacity}^{OMP} = T_{noOMP} + T_{serial}^{OMP} + T_{Imbal}^{OMP} + T_{sched}^{OMP}$$

and OpenMP Parallel Efficiency is:

$$OMP_PE = \frac{T_{noOMP}}{T_{noOMP} + T_{serial}^{OMP} + T_{Imbal}^{OMP} + T_{sched}^{OMP}}$$

OpenMP Serial Efficiency

OpenMP Serial Efficiency characterizes the loss of thread capacity caused by execution outside OpenMP parallel regions.

BasicAnalysis does not interpret the duration of a serial region itself as the complete loss. Instead, it estimates the capacity that remains unused because the other OpenMP threads cannot contribute useful parallel work during that interval.

For each MPI rank in an MPI+OpenMP execution, or each OpenMP process in an OpenMP-only execution, BasicAnalysis first determines the duration for which OpenMP parallelism is not active.

Let r denote an MPI rank for an MPI+OpenMP execution or an OpenMP process for an OpenMP-only execution.

Symbol	Description
$T_{serial,r}^{active}$	Duration for which OpenMP parallelism is not active for r .
$N_{threads,r}$	Number of OpenMP threads associated with r .

The corresponding lost thread capacity is:

$$T_{serial,r}^{OMP} = T_{serial,r}^{active} \cdot (N_{threads,r} - 1)$$

and the total serial loss is:

$$T_{serial}^{OMP} = \sum_r T_{serial,r}^{OMP}$$

For example, if an OpenMP process with four threads spends 10 ms outside parallel regions, the serial-region duration is 10 ms, but the corresponding lost worker-thread capacity is:

$$10\text{ ms} \cdot (4 - 1) = 30\text{ ms}$$

This distinction is important because OpenMP Serial Efficiency characterizes the loss of available parallel capacity, rather than simply the elapsed duration of serial regions.

BasicAnalysis computes:

$$OMP_Serial = \frac{T_{noOMP}}{T_{noOMP} + T_{serial}^{OMP}}$$

A value close to 100% indicates that little thread capacity is lost because OpenMP parallelism is inactive. Lower values indicate an increasing contribution from serial execution outside OpenMP parallel regions.

OpenMP Load Balance

OpenMP Load Balance characterizes how evenly useful computation is distributed among OpenMP threads inside parallel regions.

For each OpenMP parallel region p , let:

$$U_{p,i}$$

be the useful-computation duration of thread i , and:

$$U_p^{max} = \max_i(U_{p,i})$$

the maximum useful-computation duration among the threads participating in that region.

BasicAnalysis estimates the thread-capacity loss associated with useful-work imbalance as:

$$T_{Imbal,p}^{OMP} = \sum_i (U_p^{max} - U_{p,i})$$

The imbalance contribution is then accumulated over the OpenMP parallel regions represented in the trace:

$$T_{Imbal}^{OMP} = \sum_p T_{Imbal,p}^{OMP}$$

OpenMP Load Balance is computed as:

$$OMP_LB = \frac{T_{noOMP} + T_{serial}^{OMP}}{T_{noOMP} + T_{serial}^{OMP} + T_{Imbal}^{OMP}}$$

A value close to 100% indicates that useful work is similarly distributed among the OpenMP threads. Lower values indicate increasing useful-work imbalance.

Therefore, this metric measures **useful-work imbalance** rather than directly measuring time spent in an OpenMP waiting state.

OpenMP Scheduling Efficiency

OpenMP Scheduling Efficiency characterizes the OpenMP runtime overhead associated with scheduling and fork/join activity, after serial-execution and useful-work-imbalance losses have been represented separately in the multiplicative decomposition.

The T_{sched}^{OMP} component defined above represents this remaining OpenMP runtime-related thread-capacity loss.

BasicAnalysis computes:

$$OMP_Sched = \frac{T_{noOMP} + T_{serial}^{OMP} + T_{Imbal}^{OMP}}{T_{noOMP} + T_{serial}^{OMP} + T_{Imbal}^{OMP} + T_{sched}^{OMP}}$$

A value close to 100% indicates that little thread capacity is lost to the remaining OpenMP runtime overhead. Lower values indicate an increasing contribution from scheduling and fork/join overhead.

Multiplicative decomposition

The three OpenMP runtime-specific factors form a multiplicative decomposition of OpenMP Parallel Efficiency:

$$OMP_PE = OMP_Serial \cdot OMP_LB \cdot OMP_Sched$$

Substituting the individual definitions gives:

$$OMP_PE = \frac{T_{noOMP}}{T_{noOMP} + T_{serial}^{OMP}} \cdot \frac{T_{noOMP} + T_{serial}^{OMP}}{T_{noOMP} + T_{serial}^{OMP} + T_{Imbal}^{OMP}} \cdot \frac{T_{noOMP} + T_{serial}^{OMP} + T_{Imbal}^{OMP}}{T_{noOMP} + T_{serial}^{OMP} + T_{Imbal}^{OMP} + T_{sched}^{OMP}}$$

which reduces to:

$$OMP_PE = \frac{T_{noOMP}}{T_{noOMP} + T_{serial}^{OMP} + T_{Imbal}^{OMP} + T_{sched}^{OMP}}$$

This decomposition allows a reduction in OpenMP Parallel Efficiency to be attributed to serial execution, useful-work imbalance, or the remaining OpenMP runtime scheduling and fork/join overhead.

6.5.2 OpenMP-Only and MPI+OpenMP Applications

The OpenMP Runtime-Specific metrics are available for both **OpenMP-only** and **MPI+OpenMP** applications. The same OpenMP efficiency decomposition is used in both cases, but the quantities used to construct the model differ according to the programming model.

In particular, the difference affects T_{noOMP} and the active master-thread time used to compute the serial-loss component.

For an OpenMP-only application:

$$T_{noOMP} = T_{useful}$$

For an MPI+OpenMP application:

$$T_{noOMP} = T_{useful} + T_{MPI}$$

MPI execution is included in T_{noOMP} because the OpenMP Runtime-Specific metrics are designed to isolate inefficiencies attributable to the OpenMP runtime. Time spent executing MPI calls is therefore not classified as an OpenMP runtime loss.

MPI execution can nevertheless contribute indirectly to the OpenMP serial-loss component through unused worker-thread capacity. When the master thread executes an MPI call outside an OpenMP parallel region, the remaining OpenMP worker threads are inactive. The corresponding unused worker-thread capacity contributes to T_{serial}^{OMP} .

This treatment allows the same OpenMP efficiency decomposition to be applied to OpenMP-only and MPI+OpenMP executions while preserving the runtime-specific interpretation of the metrics.

6.5.3 Interpretation

The OpenMP Runtime-Specific metrics distinguish the main sources of OpenMP-related efficiency loss:

- A low OMP_Serial indicates a significant loss of OpenMP thread capacity while OpenMP parallelism is not active.
- A low OMP_LB indicates uneven useful-work distribution among OpenMP threads inside parallel regions.
- A low OMP_Sched indicates a significant contribution from the remaining OpenMP runtime scheduling and fork/join overhead.
- A low OMP_PE indicates that one or more of these OpenMP-specific factors are limiting the efficiency of the OpenMP execution.

For MPI+OpenMP applications, this runtime-specific view complements the OpenMP contribution reported by the Parallel Runtime Model. The two views address different performance questions:

- The **Parallel Runtime Model** quantifies the OpenMP contribution within the application-level multiplicative efficiency decomposition.
- The **OpenMP Runtime-Specific Analysis** characterizes the OpenMP execution itself and identifies which OpenMP-specific factor contributes to its inefficiency.

Using both views is particularly useful for hybrid applications because the runtime-specific analysis can expose OpenMP behavior that may not be apparent from the application-level runtime decomposition alone.

6.6 I/O Metrics

BasicAnalysis provides complementary metrics for characterizing File I/O activity observed in the trace.

These metrics are reported independently from the hierarchical performance-efficiency model. They do not currently contribute to Global Efficiency, Parallel Efficiency, Computation Scalability, or the runtime-specific efficiency decompositions.

BasicAnalysis distinguishes between:

- MPI-I/O activity; and

- POSIX and ANSI C File I/O activity.

API-specific I/O metrics are available when the corresponding I/O activity is present in the trace.

6.6.1 I/O Efficiency

I/O Efficiency characterizes the overall weight of File I/O activity relative to useful computation.

The following notation is used by the I/O metrics:

Symbol	Description
<i>Useful</i>	Total useful computation time.
<i>MPIIO</i>	Total I/O time done by MPI-I/O calls measured in the trace.
<i>POSIXIO</i>	Total I/O time done by POSIX and ANSI C File I/O calls measured in the trace.
<i>P</i>	Number of Host execution units considered by the I/O metric. For simple execution models, this corresponds to the parallel-unit count of the analyzed configuration. For hybrid applications, accelerator execution contexts are excluded because File I/O activity is attributed to the Host.
<i>T</i>	Application execution time represented by the analyzed trace.

BasicAnalysis computes:

$$IO_Eff = \frac{Useful}{Useful + MPIIO + POSIXIO}$$

A value close to 100% indicates that File I/O represents only a small fraction of the activity considered by this metric relative to useful computation. A lower value indicates that File I/O has an increasing weight relative to useful computation.

This metric should not be interpreted as a measure of storage-system performance or I/O bandwidth. It characterizes the relative contribution of the File I/O activity observed in the trace.

6.6.2 MPI I/O Efficiency

MPI I/O Efficiency characterizes the fraction of the aggregate execution capacity that is not spent in MPI-I/O activity.

BasicAnalysis computes:

$$MPI_IO_Eff = 1 - \frac{MPIIO}{PT}$$

where *MPIIO* is the total MPI-I/O duration accumulated across the parallel execution units represented in the measurement.

A value close to 100% indicates that MPI-I/O consumes only a small fraction of the aggregate execution capacity. Lower values indicate an increasing contribution from MPI-I/O activity.

MPI I/O Efficiency is reported as `Non-Avail` when no MPI-I/O activity is detected in the analyzed execution.

6.6.3 MPI I/O Load Balance

MPI I/O Load Balance characterizes how evenly MPI-I/O activity is distributed across the execution units represented in the MPI-I/O measurement.

Let *MPIIO_i* be the MPI-I/O duration associated with execution unit *i*. BasicAnalysis computes:

$$MPI_IO_LB = \frac{\overline{MPIIO}}{\max(MPIIO)}$$

where $\overline{MPIIO}$ is the average MPI-I/O duration across the execution units represented in the measurement.

A value close to 100% indicates that MPI-I/O duration is similarly distributed across those execution units. Lower values indicate that some execution units spend substantially more time in MPI-I/O than others.

A low MPI I/O Load Balance may be associated with uneven I/O volumes, rank-specific I/O behavior, serialized I/O, or aggregator-dominated access patterns. Detailed trace or I/O analysis is required to determine the underlying cause.

MPI I/O Load Balance is reported as `Non-Avail` when no MPI-I/O activity is detected.

6.6.4 POSIX I/O Efficiency

POSIX I/O Efficiency characterizes the fraction of the aggregate execution capacity that is not spent in POSIX or ANSI C File I/O activity captured by the trace.

BasicAnalysis computes:

$$POSIX_IO_Eff = 1 - \frac{POSIXIO}{PT}$$

where $POSIXIO$ is the total POSIX and ANSI C File I/O duration accumulated across the parallel execution units represented in the measurement.

A value close to 100% indicates that POSIX and ANSI C File I/O consumes only a small fraction of the aggregate execution capacity. Lower values indicate an increasing contribution from this File I/O activity.

POSIX I/O Efficiency is reported as `Non-Avail` when no POSIX or ANSI C File I/O activity is detected in the analyzed execution.

6.6.5 POSIX I/O Load Balance

POSIX I/O Load Balance characterizes how evenly POSIX and ANSI C File I/O activity is distributed across the execution units represented in the measurement.

Let $POSIXIO_i$ be the File I/O duration associated with execution unit i . BasicAnalysis computes:

$$POSIX_IO_LB = \frac{\overline{POSIXIO}}{\max(POSIXIO)}$$

where $\overline{POSIXIO}$ is the average POSIX and ANSI C File I/O duration across the execution units represented in the measurement.

A value close to 100% indicates that File I/O duration is similarly distributed across those execution units. Lower values indicate increasing imbalance, with some execution units spending substantially more time in File I/O than others.

A low POSIX I/O Load Balance may result from uneven data distribution, rank-specific File I/O, serialized access, or differences in the amount of data processed by individual execution units. Detailed trace or I/O analysis is required to determine the underlying cause.

POSIX I/O Load Balance is reported as `Non-Avail` when no POSIX or ANSI C File I/O activity is detected.

6.6.6 Interpretation of the I/O Metrics

The I/O metrics provide complementary perspectives on the File I/O activity observed during execution.

I/O Efficiency characterizes the overall weight of File I/O relative to useful computation. In contrast, **MPI I/O Efficiency** and **POSIX I/O Efficiency** characterize how much of the aggregate execution capacity is not consumed by activity associated with the corresponding File I/O interface. The API-specific efficiencies are therefore not simply a decomposition of I/O Efficiency.

The I/O Load Balance metrics provide a different perspective by characterizing the distribution of I/O duration among the execution units represented in the corresponding measurement. They can therefore reveal I/O imbalance even when the overall weight of I/O is relatively small.

Conversely, a high I/O Load Balance does not imply that File I/O overhead is small. All execution units may spend a similarly large amount of time in I/O. The efficiency and load-balance metrics should therefore be interpreted together.

These metrics quantify the time contribution and distribution of the File I/O activity captured in the trace. They do not directly characterize transferred data volume, achieved bandwidth, storage-system utilization, or the efficiency of individual I/O operations. When File I/O represents a significant performance factor, additional trace analysis or specialized I/O analysis may be required to identify the underlying cause.

6.7 General Metrics

BasicAnalysis also reports the classic parallel-performance indicators **Speedup** and **Scaling Efficiency** when multiple execution configurations are analyzed.

These quantities describe the overall scaling behavior of the application and are computed with respect to the reference execution. They are general scaling indicators and do not form part of the multiplicative performance-efficiency model described in the preceding sections.

Let:

Symbol	Description
$T(0)$	Execution time of the reference configuration.
$T(n)$	Execution time of configuration n .
$P(0)$	Application-level parallel-unit count of the reference configuration.
$P(n)$	Application-level parallel-unit count of configuration n .

The parallel-unit count is the application-level execution-unit count used by BasicAnalysis to represent each configuration in the overall scaling experiment. It should not be confused with the Host execution-unit count or the number of physical devices used by the corresponding execution-domain scalability metrics.

6.7.1 Speedup

For strong scaling, BasicAnalysis computes:

$$Speedup(n) = \frac{T(0)}{T(n)}$$

Under ideal strong scaling, Speedup grows proportionally with the increase in parallel resources.

For weak scaling, BasicAnalysis computes:

$$Speedup(n) = \frac{T(0)}{T(n)} \cdot \frac{P(n)}{P(0)}$$

The resource ratio accounts for the increase in workload expected in a weak-scaling experiment. This normalization allows the reported Speedup to retain the resource-growth interpretation: under ideal weak scaling, execution time remains constant while workload and resources increase proportionally, and Speedup therefore follows the resource-growth ratio.

6.7.2 Scaling Efficiency

Scaling Efficiency expresses the achieved scaling behavior relative to the ideal behavior defined by the selected scaling model.

For strong scaling, BasicAnalysis computes:

$$Efficiency(n) = \frac{T(0)}{T(n)} \cdot \frac{P(0)}{P(n)}$$

or equivalently:

$$Efficiency(n) = \frac{Speedup(n)}{P(n)/P(0)}$$

For weak scaling, BasicAnalysis computes:

$$Efficiency(n) = \frac{T(0)}{T(n)}$$

For both scaling models, a value of 1 represents ideal scaling behavior relative to the reference execution. Values below 1 indicate increasing loss of scaling efficiency, while values above 1 indicate scaling behavior better than the corresponding ideal reference, for example in a superlinear strong-scaling case.

Unlike the efficiency factors of the hierarchical performance model, which are reported as percentages, Scaling Efficiency is reported as a dimensionless value, where 1 corresponds to 100% efficiency.

6.8 Metric Availability

Some BasicAnalysis metrics require information that may not be available for every trace or analysis configuration.

BasicAnalysis reports `Non-Avail` when the information required to compute a metric is not available or when the metric is not defined for the corresponding analysis.

Typical cases include:

- application-level Computation Scalability and its associated scalability factors when only one execution configuration is analyzed;
- IPC Scalability, Instruction Scalability, and Frequency Scalability, including their Host-side counterparts when applicable, when the required hardware-counter information is not available;
- Host and Device Computation Scalability when only one MPI+GPU execution configuration is analyzed;
- MPI Serialization Efficiency and MPI Transfer Efficiency when the required Dimemas simulation is unavailable, explicitly skipped, or unsupported for the detected programming model;
- derived OpenMP Serialization and Transfer contributions when the optional MPI+OpenMP hybrid simulation is not requested or cannot be completed;
- MPI I/O Efficiency and MPI I/O Load Balance when no MPI-I/O activity is detected;
- POSIX I/O Efficiency and POSIX I/O Load Balance when no POSIX or ANSI C File I/O activity is detected; and
- metrics that are not defined for the detected trace mode or programming model.

A value reported as `Non-Avail` therefore indicates that BasicAnalysis cannot evaluate that metric from the information available for the current analysis. It should not be interpreted as zero efficiency.

The set of available metrics depends on the trace mode, programming model, number of analyzed execution configurations, available hardware counters, detected runtime and I/O activity, availability of the required simulation, and the selected analysis options.

PERFORMANCE REPORT

BasicAnalysis generates an interactive performance report that organizes the analysis into complementary views. Each view addresses a different question about the application's performance and helps progressively narrow down the factors responsible for efficiency loss.

The report is intended to support a hierarchical analysis workflow. Start with the Execution Overview to verify the analyzed configurations, then use the Parallel Runtime Model to identify the main sources of efficiency loss. Runtime-Specific Analysis provides a more detailed view of the active parallel runtimes, while Execution Domains identifies where accelerator-related inefficiencies manifest. I/O Analysis provides a complementary view of the weight and distribution of File I/O activity. Finally, the Scaling view shows how performance and efficiency factors evolve across the analyzed configurations.

Throughout this workflow, Metric Details provides contextual guidance to help interpret the reported metrics and continue the performance diagnosis.

These views are complementary and may use different analytical scopes. Not all metrics shown in the report belong to the same multiplicative efficiency model.

The following sections describe the report interface, the purpose of each analysis view, and the functionality available to support the performance analysis.

7.1 Report Interface

The BasicAnalysis performance report is organized around five primary analysis views:

- **Execution Overview**
- **Parallel Runtime Model**
- **Execution Domains**
- **I/O Analysis**
- **Scaling**

Use the navigation bar shown in [Fig. 7.1](#) to move between the different report views.

Runtime-specific analysis is accessed from the Parallel Runtime Model through the **Runtime Analysis** selector rather than as a separate primary view.

Metric values in the interactive efficiency tables can be clicked to open the **Metric Details** view. This view provides the metric definition, interpretation of the observed value, and recommended next diagnostic steps. The Metric Details functionality is described later in this chapter.

The controls on the right side of the navigation bar provide access to the complementary split view and to the report export functions. These features are also described later in this chapter.

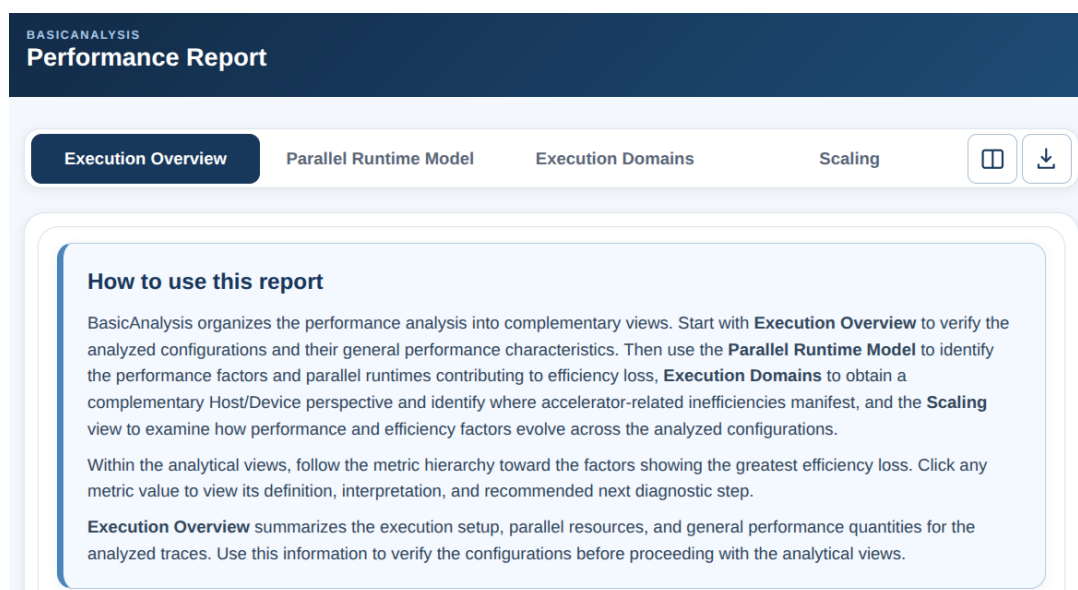


Fig. 7.1: Performance report navigation.

7.2 Execution Overview

The **Execution Overview**, shown in Fig. 7.2, is the starting point of the analysis. It summarizes the execution configuration and general performance characteristics of the analyzed traces.

Trace configuration

ID	Trace	Trace mode	Programming model	Parallel units	MPI ranks	Streams/rank
T1	cuda-cloverleaf_4p+4d.prv	Detailed	MPI + CUDA	8	4	
T2	cuda-cloverleaf_8p+8d.prv	Detailed	MPI + CUDA	16	8	
T3	cuda-cloverleaf_16p+16d.prv	Detailed	MPI + CUDA	32	16	

General metrics

Trace columns: Parallel units (MPI ranks × streams/rank) [nD = devices]

Metric	8 (4×1) [4D]	16 (8×1) [8D]	32 (16×1) [16D]
Runtime			
Elapsed time (s)	3.38	2.25	2.35

Fig. 7.2: Execution Overview.

Before interpreting efficiency metrics, verify that the reported trace configuration corresponds to the intended experiment. In particular, check the programming model, parallel resources, and accelerator configuration when applicable.

The General Metrics table provides the main execution quantities used to compare the analyzed configurations, including elapsed time, Speedup, Efficiency, IPC, and processor frequency when available.

When several execution configurations are analyzed, these values provide an initial indication of whether application performance improves as resources increase. The detailed efficiency and scaling views should then be used to identify which factors explain the observed behavior.

7.3 Parallel Runtime Model

The **Parallel Runtime Model**, shown in Fig. 7.3, identifies the performance factors and parallel runtimes contributing to efficiency loss.

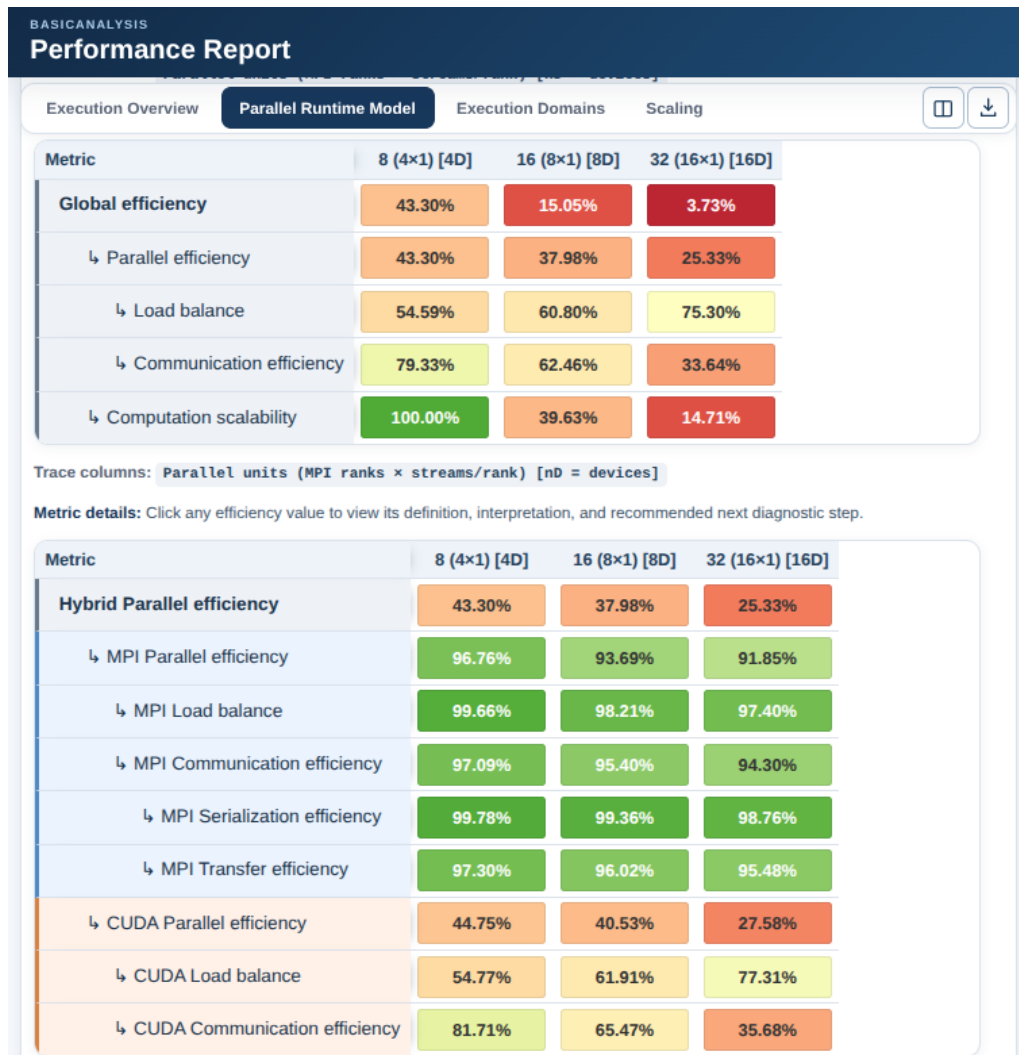


Fig. 7.3: Parallel Runtime Model view.

Start with **Global Efficiency** and follow its child metrics toward the lowest-efficiency factors. Global Efficiency separates losses associated with Parallel Efficiency from changes in Computation Scalability.

For hybrid applications, the Parallel Runtime Model additionally decomposes Parallel Efficiency across the active parallel runtimes.

The model can therefore be read in two complementary ways:

- **by performance factor**, following Global Efficiency toward Parallel Efficiency, Load Balance, Communication Efficiency, and Computation Scalability; or
- **by runtime**, comparing the contributions of MPI and the inner parallel runtime.

The performance-factor hierarchy identifies what type of efficiency loss is present, while the runtime decomposition helps determine which parallel runtime contributes to the observed loss.

In the interactive report, metric values are selectable. Clicking a metric value opens its definition, interpretation, and recommended next diagnostic step.

7.4 Runtime-Specific Analysis

When a runtime requires closer investigation, select it from the **Runtime Analysis** selector. A split view opens, displaying the metrics specific to the selected runtime while keeping the Parallel Runtime Model visible for context. An example for the OpenMP parallel runtime is shown in Fig. 7.4.

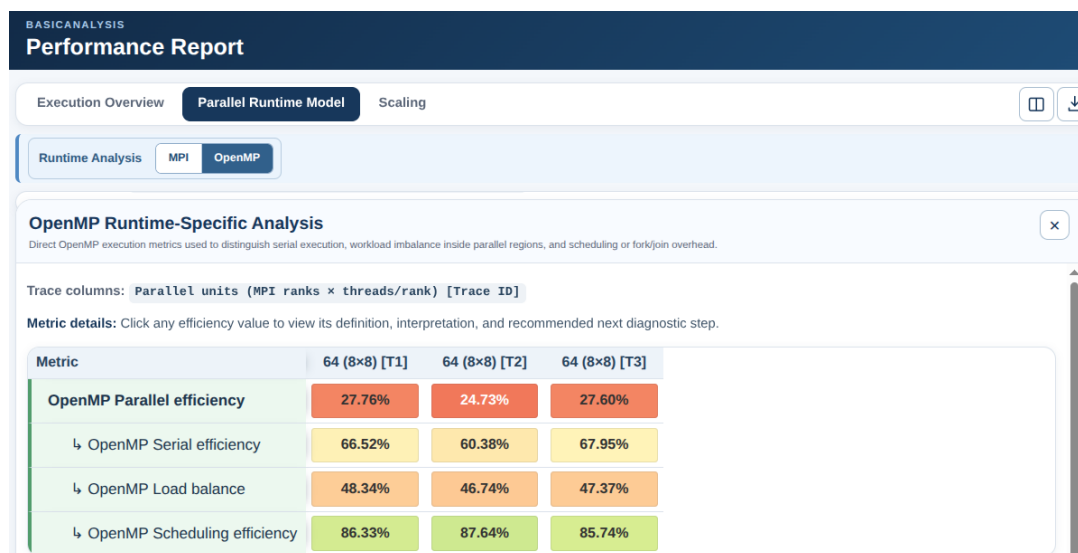


Fig. 7.4: Runtime-specific analysis.

The selected runtime opens in the detailed-analysis panel while the Parallel Runtime Model remains visible for context.

The available runtime-specific metrics depend on the programming model. For example:

- MPI analysis examines MPI Load Balance and Communication Efficiency, including Serialization and Transfer Efficiency when available.
- OpenMP analysis provides the isolated OpenMP Serial, Load Balance, and Scheduling efficiencies.
- CUDA or HIP analysis displays the corresponding accelerator-runtime contribution defined by the selected hybrid model.

Runtime-Specific Analysis is intended to explain the runtime contribution identified in the Parallel Runtime Model. Its metrics may therefore use a different analytical scope from the application-level metrics.

7.5 Execution Domains

For accelerator applications, the **Execution Domains** view provides a complementary analysis of where performance inefficiencies manifest by separating the execution into **Host** and **Device** domains.

Fig. 7.5 shows an example of the Host and Device execution-domain analysis.

The Host domain characterizes CPU-side execution, including Host Parallel Efficiency, accelerator offload behavior, and Host Computation Scalability.

The Device domain characterizes accelerator execution through Device Parallel Efficiency and its Load Balance, Communication, and Orchestration components, together with Device Computation Scalability.

For Device metrics, GPU stream activity is first mapped and flattened to the corresponding physical devices. Consequently, the Device analysis represents physical accelerator activity rather than treating individual GPU streams as independent parallel execution units.

Host and Device are complementary execution domains and are **not** components of a common multiplicative efficiency model. Therefore, Host Global Efficiency and Device Global Efficiency must be interpreted independently and must not be multiplied to obtain the application Global Efficiency.

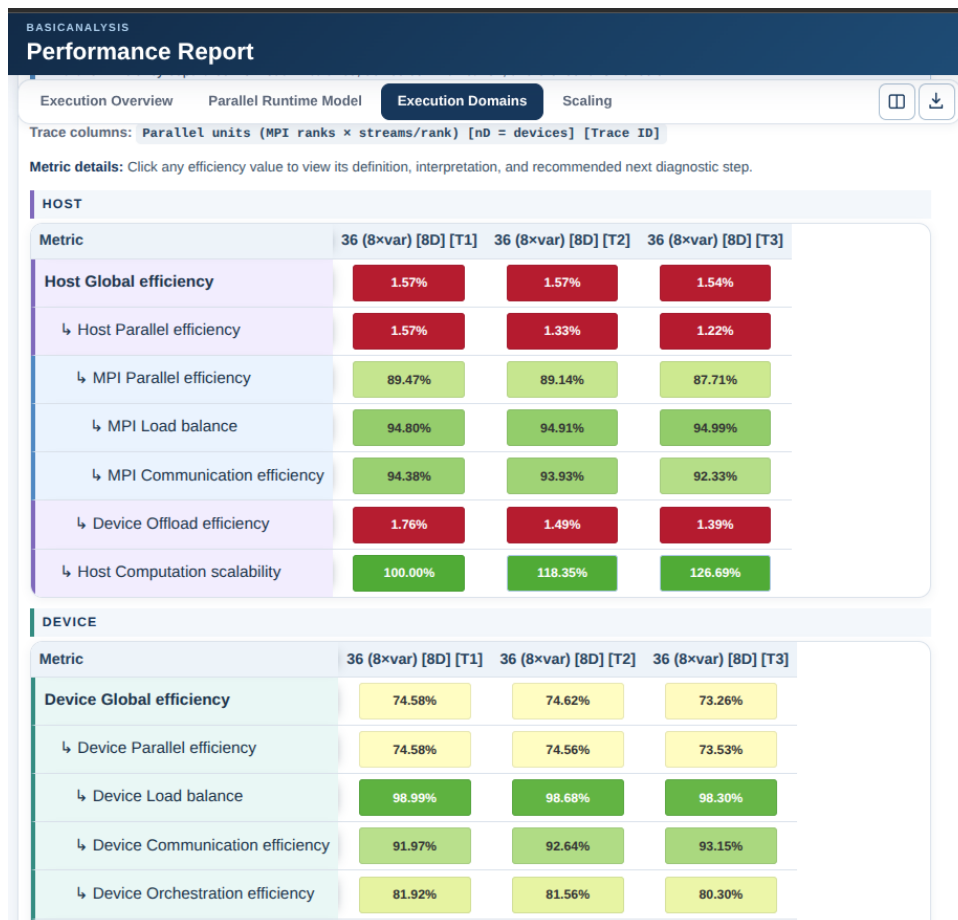


Fig. 7.5: Host and Device execution-domain analysis.

Use the Parallel Runtime Model to determine **which runtime contributes to the efficiency loss**, and the Execution Domains view to determine **where accelerator-related inefficiencies manifest**.

A low execution-domain metric identifies where a performance loss becomes visible. Detailed runtime or trace analysis may still be required to determine the underlying cause.

7.6 I/O Analysis

The **I/O Analysis** view provides a complementary characterization of the File I/O activity captured in the analyzed traces.

BasicAnalysis distinguishes between **MPI-I/O** and **POSIX/ANSI C File I/O** activity and reports the corresponding metrics when that activity is detected.

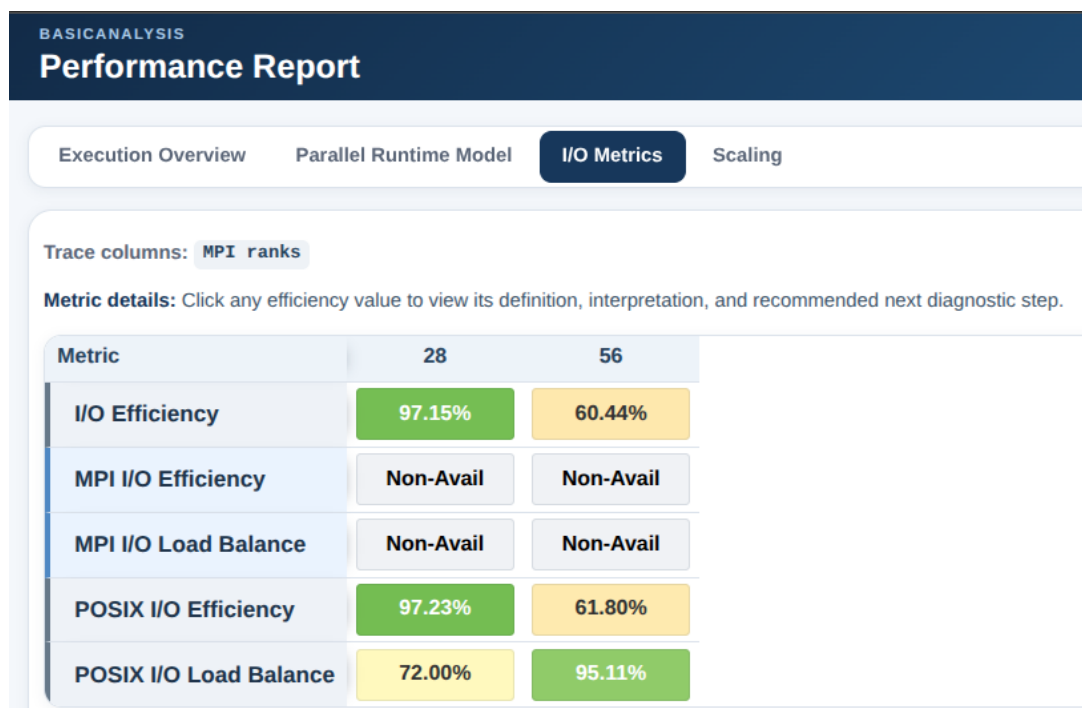


Fig. 7.6: I/O Analysis.

The **I/O Analysis** view is shown in Fig. 7.6. This includes:

- **I/O Efficiency**, which characterizes the overall weight of File I/O activity relative to useful computation;
- **MPI I/O Efficiency**, which characterizes the fraction of aggregate execution capacity not spent in MPI-I/O activity;
- **MPI I/O Load Balance**, which characterizes how evenly MPI-I/O duration is distributed across the execution units represented in the measurement;
- **POSIX I/O Efficiency**, which characterizes the fraction of aggregate execution capacity not spent in POSIX or ANSI C File I/O activity; and
- **POSIX I/O Load Balance**, which characterizes how evenly that File I/O activity is distributed across the execution units represented in the measurement.

I/O Efficiency and the API-specific I/O efficiencies answer questions about the **weight of I/O activity**, whereas the Load Balance metrics characterize its **distribution**. These two aspects should be interpreted together.

For example, a high I/O Load Balance does not necessarily indicate that I/O overhead is small. All execution units may spend a similarly large amount of time performing File I/O. Conversely, File I/O may have a small overall weight while still being unevenly distributed.

When no activity for a particular File I/O interface is detected, BasicAnalysis reports the corresponding API-specific Efficiency and Load Balance metrics as **Non-Avail**.

The I/O metrics are complementary diagnostic metrics. They are reported independently from the hierarchical performance-efficiency model and must not be interpreted as additional multiplicative components of Global Efficiency, Parallel Efficiency, or Computation Scalability.

Use the I/O Analysis view when File I/O represents a relevant fraction of the execution or when the distribution of I/O activity requires further investigation. Detailed trace analysis or specialized I/O tools may be required to determine the underlying cause of an observed I/O limitation.

7.7 Scaling

The **Scaling** view analyzes how performance and efficiency factors evolve across the analyzed execution configurations.

Fig. 7.7 shows the first part of the Scaling view, which reports the detected and selected scaling model and compares the measured Speedup and Efficiency with their ideal behavior.

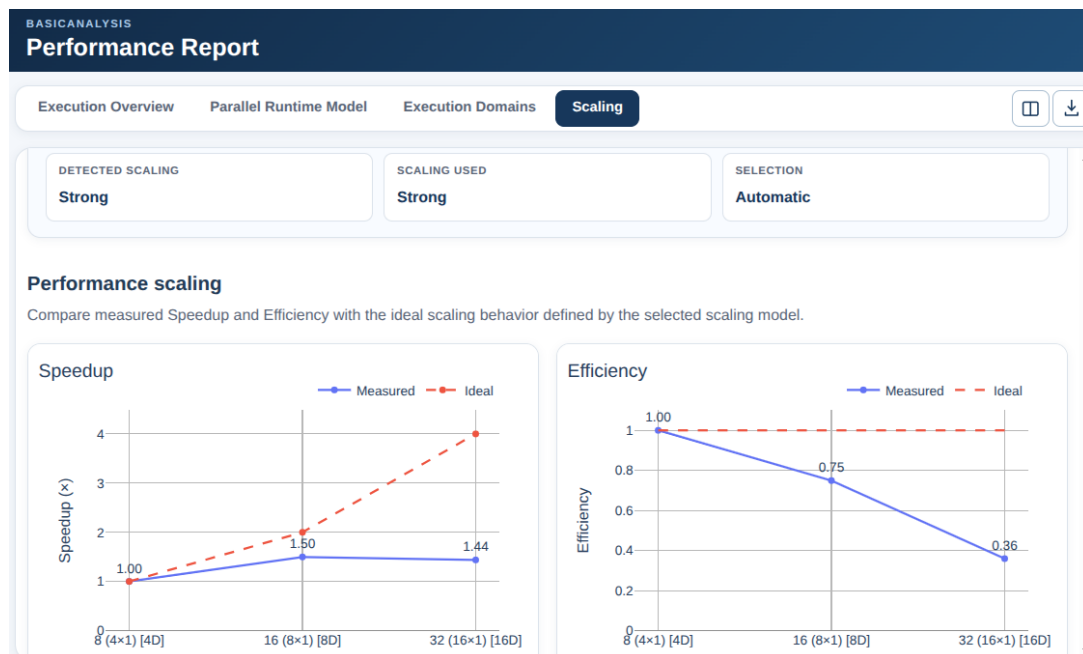


Fig. 7.7: Scaling model and performance trends.

The remaining plots show how the efficiency hierarchy and complementary runtime/domain metrics evolve across configurations. Fig. 7.8 shows an example of one of these metric trends.

Depending on the programming model and the available performance data, these can include:

- Global Efficiency, Parallel Efficiency, and Computation Scalability.
- Load Balance and Communication Efficiency.
- Computation Scalability and its available IPC, Instruction, and Frequency components.
- Runtime-specific Parallel Efficiency contributions.
- MPI Load Balance, Communication, Serialization, and Transfer Efficiency.
- Host execution-domain metrics.
- Device execution-domain metrics.
- I/O-related metrics when available.

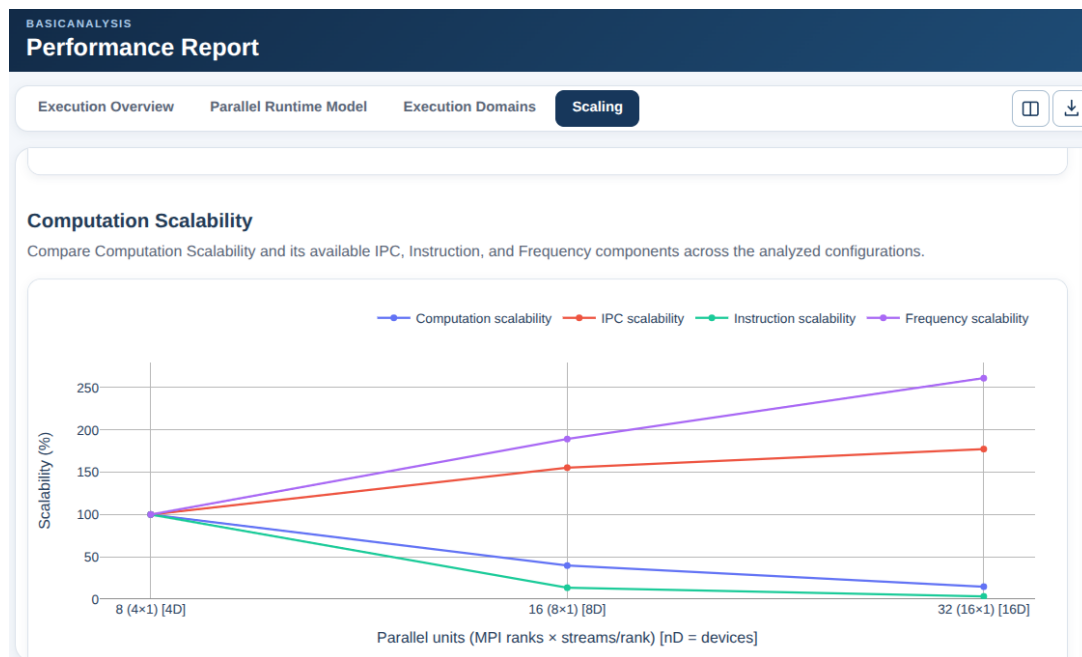


Fig. 7.8: Efficiency-factor scaling trends.

Computation Scalability and its associated factors are evaluated relative to a reference execution. A value of 100% indicates behavior equivalent to the reference under the selected scaling model, values below 100% indicate degradation, and values above 100% indicate improvement.

Other efficiency metrics shown in the Scaling view retain their own metric definitions. Their trends should be interpreted by examining how the metric value changes across configurations rather than assuming that every plotted metric is normalized to the reference execution.

Use the trends to identify which factors degrade as the application scales. A degrading metric identifies where the scalability loss becomes visible, but it does not by itself establish the root cause. Further trace analysis may be required to explain the observed behavior.

7.8 Metric Details

The interactive report provides additional information for each performance metric through the **Metric Details** view. This information helps users understand the meaning of a metric value and identify the next steps for further performance analysis.

To open the Metric Details view, click a metric value in any of the interactive efficiency tables. A dialog is displayed for the selected metric and execution, as shown Fig. 7.9.

The dialog identifies the metric, its analysis context, the selected execution, and the corresponding metric value. It then provides:

- **Definition** – explains what the metric measures from a performance-analysis perspective.
- **Interpretation** – explains how to interpret the observed value. When applicable, the interpretation also accounts for the efficiency range associated with the value (for example, critical or low efficiency).
- **Next diagnostic step** – recommends the subsequent analysis to investigate the performance factor represented by the metric, including related metrics or execution domains that should be examined.

The information presented in **Metric Details** is adapted to the selected metric and its analysis context. This is particularly relevant for hybrid applications, where the report distinguishes between application-level metrics, contributions from the different parallel runtimes, and Host/Device execution-domain metrics.

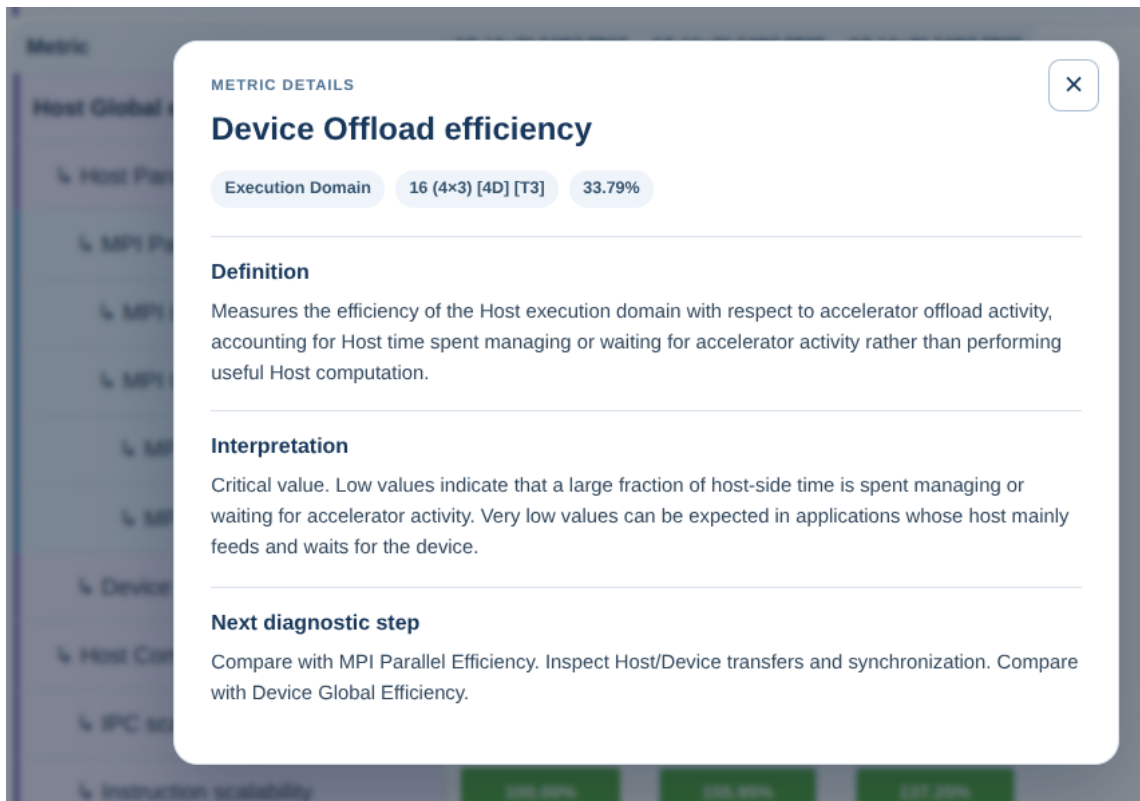


Fig. 7.9: Metric Details view for Device Offload Efficiency. The dialog provides the metric definition, interpretation of the selected value, and recommended next diagnostic steps.

7.9 Comparing Complementary Views

Fig. 7.10 shows the split-view option, which allows two analysis views to be displayed simultaneously.

This is useful when evidence from two analytical perspectives should be compared directly, for example:

- Parallel Runtime Model with Scaling;
- Parallel Runtime Model with Execution Domains;
- Parallel Runtime Model with I/O Analysis;
- Execution Domains with Scaling; or
- Execution Overview with a detailed analytical view.

The two views remain analytically independent. The split layout is intended to facilitate comparison, not to imply that the metrics shown in both panels belong to the same multiplicative model.

7.10 Exporting Report Content

The export control provides several ways to save analysis results.

Depending on the active analysis, efficiency tables can be exported as PNG images. Multiple tables can also be exported together, and all available efficiency tables can be saved in a ZIP archive.

Fig. 7.11 shows the export menu for saving an efficiency table from the Parallel Runtime Model view as a PNG image.

The complete report can be exported using **Print / Save as PDF**. This generates the printable report from the current analysis data and opens the browser print dialog, where the report can be saved as PDF.

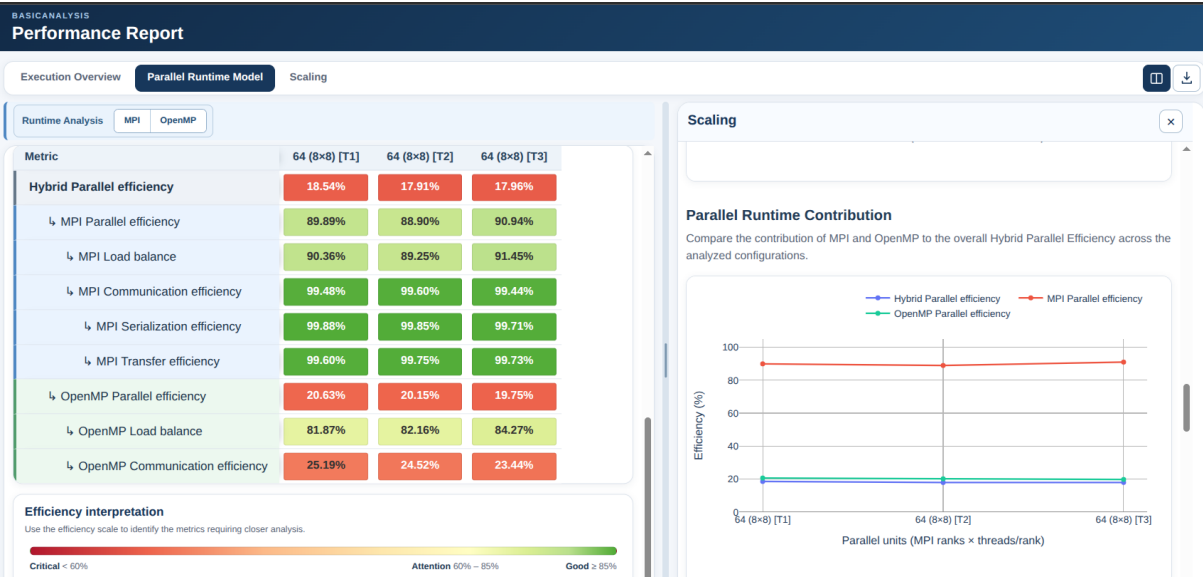


Fig. 7.10: Complementary split-view analysis.

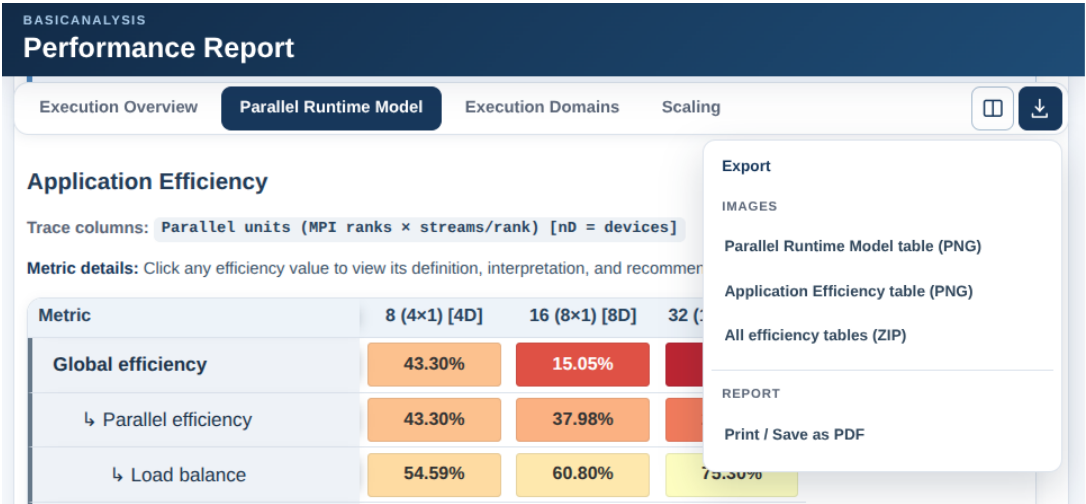


Fig. 7.11: Report export menu.

PDF generation is therefore initiated explicitly from the interactive report; BasicAnalysis does not automatically generate a PDF during normal execution.

7.11 Following the Analysis Workflow

The report views are designed to be used together rather than as independent analyses. A typical analysis follows this progression:

1. **Verify the executions** in the Execution Overview.
2. **Identify the dominant efficiency loss** using Global Efficiency and its hierarchy.
3. **Determine the responsible parallel runtime** using the Parallel Runtime Model.
4. **Explain the runtime contribution** using Runtime-Specific Analysis.
5. For accelerator applications, **identify where the inefficiency manifests** using the Host and Device execution domains.
6. When File I/O activity is relevant, **evaluate its weight and distribution** using I/O Analysis.
7. **Examine how the relevant factors evolve** across configurations in the Scaling view.
8. Use detailed trace analysis or specialized performance tools when additional evidence is required to determine the underlying cause.

This workflow is not strictly linear. The different views provide complementary evidence and may use different analytical scopes.

The Execution Overview establishes the context of the experiment. The hierarchical performance model identifies whether a significant performance loss exists and which broad factor dominates. The Parallel Runtime Model and Runtime-Specific Analysis attribute and explain parallel-runtime losses. Execution Domains localize accelerator-related inefficiencies to the Host or Device side. I/O Analysis independently characterizes the contribution and distribution of File I/O activity. Finally, the Scaling view shows how these relevant performance characteristics evolve as the execution configuration changes.

Together, these views progressively narrow the analysis from identifying a performance problem to determining which factors, runtimes, execution domains, or complementary activities should be investigated in greater detail.

OUTPUT

BasicAnalysis generates several complementary outputs from the analyzed Paraver traces. These include numerical results, static visualizations, an interactive performance report, and intermediate data produced during the analysis.

The exact files generated depend on the programming model, the available performance data, the number of analyzed traces, and the selected analysis options.

8.1 Performance Report

The main output of BasicAnalysis is the interactive performance report:

```
basicanalysis_interactive_report.html
```

The report provides a guided view of the performance analysis through the Execution Overview, Parallel Runtime Model, Runtime-Specific Analysis, Execution Domains, I/O Analysis, and Scaling views, as applicable to the analyzed execution.

The available views depend on the programming model and performance data. For example, Execution Domains are available for accelerator applications, while I/O Analysis reports the File I/O metrics derived from MPI-I/O and POSIX/ANSI C File I/O activity detected in the traces. When multiple execution configurations are analyzed, the report also provides scaling information and performance trends.

Metric values in the report are interactive. Selecting a metric opens its definition, interpretation, and recommended next diagnostic steps.

Efficiency tables can be exported from the report as PNG images. The report also provides export options for multiple tables and can be printed or saved as a PDF using the browser.

See *Performance Report* for a detailed description of the report, its analytical views, and its export capabilities.

8.2 Numerical Output

BasicAnalysis writes the numerical results of the analysis to CSV files in the output directory.

The main files include:

rawdata.csv

Contains the aggregated performance measurements extracted or constructed from the input traces and used to compute the performance metrics. These include timing and hardware-counter measurements and, when applicable, runtime-specific, I/O, and accelerator-related quantities.

other_metrics.csv

Contains complementary performance quantities such as elapsed time, Speedup, Efficiency, IPC, processor frequency, tracer flushing information, and I/O-related metrics when available.

modelfactors.csv

Contains the application-level hierarchical efficiency and scalability factors computed by BasicAnalysis.

efficiency_table*.csv

Contains efficiency metrics arranged in the format used to generate the corresponding static efficiency tables. The exact filename depends on the performance model being analyzed.

Additional model-specific CSV files may also be generated. For example, OpenMP analyses can generate `omp_talp_metrics.csv`, while MPI+GPU analyses using the Host/Device execution-domain model can generate `talp_metrics.csv`.

The availability of individual files and metrics depends on the programming model, the information contained in the traces, the number of analyzed execution configurations, and the selected analysis options.

8.3 Static Visualizations

BasicAnalysis generates static visualizations of the performance metrics when the required Python plotting dependencies are available.

Efficiency tables are generated as both PNG and PDF files. Depending on the performance model, the output includes files such as:

```
efficiency_table-matplot.png
efficiency_table-matplot.pdf
```

or the corresponding global, hybrid, runtime-specific, or Host/Device variants.

When multiple execution configurations are analyzed, BasicAnalysis also generates plots showing the evolution of performance metrics across the configurations. These include Speedup and Efficiency plots and, depending on the programming model and available data, plots for:

- Global Efficiency and Parallel Efficiency;
- Load Balance and Communication Efficiency;
- Computation Scalability and its available factors;
- runtime-specific efficiency factors; and
- Host and Device execution-domain metrics.

Some multi-configuration analyses also generate Gnuplot `.gp` files containing plotting instructions and the corresponding metric data.

I/O metric plots are provided within the interactive report and are not generated as separate static visualization files.

8.4 Intermediate Analysis Files

BasicAnalysis stores the intermediate data produced while analyzing each trace under:

```
scratch_out_basicanalysis/
```

A separate subdirectory is created for each analyzed trace. These directories contain the Paraver/paramedir statistics used to derive timing, hardware-counter, I/O, runtime-specific, and accelerator measurements.

For accelerator traces, the intermediate data can also include Host, GPU-stream, and memory-transfer statistics. GPU stream activity is subsequently mapped and flattened to physical devices when constructing the Device measurements used by the performance metrics.

OpenMP analyses include additional timing data used to characterize serial execution, parallel-region load balance, and scheduling and fork/join overhead.

When Dimemas simulation is applicable, Dimemas is available, and simulation has not been disabled, the corresponding trace directory also contains the simulation input, configuration, simulated trace, and paramedir statistics obtained from the simulated execution.

These files represent intermediate analysis data. The summarized measurements and computed metrics intended for direct inspection are provided separately by the CSV files in the main output directory.

LIMITATIONS

This section summarizes the main limitations of the current BasicAnalysis methodology and implementation. Metric-specific availability conditions are described in *Performance Metrics*.

9.1 Simulation-Derived Metrics

Serialization Efficiency and Transfer Efficiency rely on an ideal execution simulated with *Dimemas*. These metrics are therefore available only for programming models and runtime events supported by the current BasicAnalysis and *Dimemas* simulation workflow.

BasicAnalysis currently uses *Dimemas* simulation for MPI, MPI+OpenMP, and MPI+CUDA applications.

For MPI+HIP applications, MPI Communication Efficiency can be computed from the measured execution, but Serialization Efficiency and Transfer Efficiency are reported as **Non-Avail**. The current BasicAnalysis-*Dimemas* workflow has not been validated for HIP accelerator events, and BasicAnalysis therefore does not use simulation-derived communication submetrics for these executions.

Simulation-derived metrics are also unavailable when *Dimemas* is not installed, when the required simulation cannot be completed, or when simulation is explicitly disabled with the `--skip-simulation` option.

9.2 Number of Parallel Runtimes

The Parallel Runtime Model has been validated for applications with up to two parallel runtime levels, such as MPI+OpenMP and MPI+GPU.

Applications containing more than two parallel runtimes may be detected in the trace, but their decomposition into runtime-specific efficiency contributions has not been validated.

For example, an application combining MPI, a CPU threading runtime, and a GPU runtime introduces three parallel runtime levels. The current Parallel Runtime Model does not provide an independently validated efficiency decomposition for all three runtime contributions.

The application-level metrics may still provide useful performance information, but the runtime-specific decomposition should not be interpreted as a complete attribution of the efficiency loss across all active runtimes.

9.3 Hierarchical Execution Model

The Parallel Runtime Model assumes a hierarchical organization of the parallel runtimes. For hybrid applications, MPI is considered the outer parallel level and the second runtime, such as OpenMP, CUDA, or HIP, the inner parallel level.

The multiplicative runtime decomposition therefore assumes that the execution of the inner parallel runtime is compatible with this hierarchical organization.

For example, in an MPI+OpenMP application, when the master thread executes MPI calls outside an OpenMP parallel region, the remaining OpenMP worker threads are considered inactive. The corresponding unused thread capacity can then be attributed to the OpenMP serial component.

Applications whose runtime behavior substantially violates the assumed hierarchical organization, for example through concurrent useful activity across runtime levels that the model treats as mutually exclusive, may not be fully represented by the runtime decomposition.

9.4 GPU Execution-Domain Analysis

The Host/Device execution-domain analysis for MPI+CUDA and MPI+HIP applications depends on the accelerator activity represented in the Paraver trace.

GPU traces can contain multiple execution streams associated with the same physical device. BasicAnalysis maps these streams to physical devices and flattens overlapping useful-computation and memory-transfer intervals before constructing the Device metrics.

Consequently, the correctness of the Device metrics depends on BasicAnalysis being able to identify the GPU streams and map them to their corresponding physical devices from the trace information.

Device activity that is not represented in the trace, or accelerator configurations for which the stream-to-device mapping cannot be established correctly, cannot be fully characterized by the Device execution-domain metrics.

9.5 I/O Analysis

The I/O metrics characterize the time contribution and distribution of the File I/O activity captured in the trace.

BasicAnalysis currently distinguishes MPI-I/O and POSIX/ANSI C File I/O activity and reports the corresponding efficiency and load-balance metrics when that activity is detected.

These metrics do not directly characterize transferred data volume, achieved I/O bandwidth, storage-system utilization, filesystem contention, or the performance of individual I/O operations. They therefore identify whether File I/O activity has a significant execution-time contribution or an imbalanced distribution, but they do not by themselves determine the underlying storage-system cause.

When I/O activity represents a significant performance factor, detailed trace analysis or specialized I/O performance-analysis tools may be required for further diagnosis.