D3.1 Report on Benchmarking and Optimisation

Document Identification			
Status	Final	**Due Date**	31/03/2019
Version	1.0	**Submission Date**	01/04/2019

Related WP	WP3	**Document Reference**	D3.1
Related Deliverable(s)	D4.1, D5.1	**Dissemination Level (*)**	PU
Lead Participant	ICCS	**Lead Author**	Nikela Papadopoulou
Contributors	ICCS, PSNC, USTUTT, BUL, PLUS, ECMWF	**Reviewers**	Michael Gienger (USTUTT)
			Lara Lopez (ATOS)

Keywords:

High Performance Computing (HPC), Benchmarking, Profiling, Scalability

Document Information

List of Contributors	
Name	Partner
Nikela Papadopoulou, Konstantinos Nikas, Petros Anastasiadis	ICCS
Marcin Lawenda	PSNC
Sergiy Gogolenko, Abhishek Abhishek	USTUTT
Derek Groen	BUL
Gregor Bankhamer	PLUS
Milana Vuckovic	ECMWF

Document History			
Ver.	Date	Change editors	Changes
0.1	31/01/2019	ICCS	Created initial document, table of contents.
0.2	06/03/2019	ICCS, ECMWF, PSNC, USTUTT	Added content in Sections 2, 3.
0.3	12/03/2019	ICCS, PLUS	Added content in Section 4, edited partners' contributions (ECMWF, PSNC, USTUTT) in Sections 2, 3.
0.4	13/03/2019	ICCS	Edited content in Section 4, added content in Section 3.
0.5	18/03/2019	ICCS, BUL	Added & reviewed content in Section 4.
0.6	19/03/2019	ICCS	Added content in Section 2, added content in Section 3.
0.7	20/03/2019	ICCS	Pre-final version for the internal review
0.8	27/03/2019	ICCS	Addressed comments from internal review.
1.0	27/03/2019	ICCS	Final version.

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable leader	Nikela Papadopoulou (ICCS)	28/03/2019
Quality manager	Marcin Lawenda (PSNC)	28/03/2019
Project Coordinator	Francisco Javier Nieto (ATOS)	29/03/2019

Table of Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation / acronym	Description
Dx.y	Deliverable number y belonging to WP x
EC	European Commission
ENS	Members’ Ensemble
HLRS	High Performance Computing Center Stuttgart
HPC	High Performance Computing
HPCG	High Performance Conjugate Gradient
HPDA	High Performance Data Analytics
HPL	High Performance LINPACK
HRES	High Resolution Weather Forecast
MPI	Message Passing Interface
VM	Virtual Machine
WP	Work Package

Executive Summary

Deliverable D3.1 defines the HiDALGO benchmarking methodology with a focus on the HiDALGO HPC infrastructure. HiDALGO's target is the definition of a generic, systematic, reproducible, and interpretable methodology for collecting benchmarking information from the HiDALGO applications, and a systematic way of storing benchmarking results. To achieve that, this deliverable study the existing HiDALGO infrastructure, surveys available tools, and draws from best practices for HPC systems and applications. It also presents the preliminary effort to apply this methodology on the HiDALGO pilot applications.

The HiDALGO methodology has been fully applied on the Migration pilot. Similar efforts have been kick-started for benchmarking the Air Pollution and Social Networks pilots as well. This initial experimentation and benchmarking have helped identify various major and minor issues in procuring and/or benchmarking the HiDALGO pilots, and has significantly impacted the definition of the HiDALGO methodology.

1 Introduction

1.1 Purpose of the document

Deliverable 3.1 is prepared in the context of WP 3, which aims to identify the issues that prevent software that deals with Global Challenges from achieving the highest possible computation performance (exactable performance), as well as efficient and agile data manipulation. This deliverable contributes to developing a benchmarking methodology to assess the quality and performance of applications and obtaining detailed information on their scalability behaviour and performance bottlenecks. This way, this deliverable contributes to the project's goals towards excellence in application design and optimization.

1.2 Relation to other project work

Deliverable 3.1 summarises initial findings regarding the performance and scalability of HiDALGO applications, as these are described in Deliverable 4.1 (*Initial Status of the Pilot Applications*), on the project's HPC infrastructure, as described in Deliverable 5.1 (*HiDALGO System Environment*). It drives future activities within WP 3 (*Benchmarking and Co-design, Scalability, Coupling*) and within WP 4 (*Implementation of the Pilot applications*). It is the first of a series of reports focusing on the benchmarking, implementation, optimisation, and coupling technologies (D3.3, D3.4, and D3.5).

1.3 Structure of the document

The document is structured into 3 major chapters.

Chapter 2 presents the HiDALGO Benchmarking methodology. It describes existing infrastructure, lists available tools for performance analysis, sets the aims and goals of the benchmarking methodology and provides a description of the workflow for benchmarking HiDALGO applications.

Chapter 3 presents the current status of deploying the HiDALGO Pilot applications on the project's HPC infrastructure together with initial findings on their scalability and profiling results.

Chapter 4 concludes the document summarising key findings for each application and lessons learned from this first initial effort towards setting up and implementing the HiDALGO benchmarking methodology. Finally, it sets the next goals for benchmarking within HiDALGO.

2 HiDALGO Benchmarking

2.1 HiDALGO HPC/HPDA infrastructure

This section describes the infrastructure used for benchmarking in the context of this deliverable. A more detailed description of the existing HiDALGO infrastructure can be found in Deliverable 5.1 of WP 5. As the activities of Task 3.1 continue, benchmarking will be extended to any infrastructure that will be made available to the project in the future.

2.1.1 USTUTT infrastructure

For Deliverable 3.1, we have utilized the Cray XC-40 Hazelhen, the flagship system at HLRS. The system is a homogeneous supercomputer, built upon Intel Haswell Processors, interconnected via the proprietary Cray Aries network in the dragonfly topology. Each node consists of two 24-core processors and 128 GB of memory. The system ranks 20th on the Top500 list and provides a peak performance of 7.4 PFlop/s. More information can be found in Deliverable 5.1.

2.1.2 PSNC infrastructure

For Deliverable 3.1, we have utilized the CPU partition of Eagle, the supercomputer at PSNC, which is based on Intel Haswell processors. Each node consists of two CPUs of 14 or 16 cores each, with the memory per node varying between 64 GB and 256 GB. The nodes in Eagle are interconnected with InfiniBand FDR. More information can be found in Deliverable 5.1.

2.1.3 ECMWF infrastructure

ECMWF will not provide direct access to its HPC facilities in the context of HiDALGO. Instead, ECMWF is developing cloud environments co-hosted to the HPC, which will offer very fast and easy access to HPC resources, such as disks. The description of ECMWF infrastructure and weather forecast model used for benchmarking can be found in Deliverable 5.1.

2.2 HiDALGO application analysis tools

2.2.1 HiDALGO application analysis targets

The two systems described in Section 2.1, i.e. Hazelhen and Eagle, facilitate the performance analysis of running applications with a wide set of tools. We summarise the list of pre-installed tools in Table 2.1. Some of the tools are available on both HLRS and PSNC infrastructures, some come with a particular license and are installed on one of the two systems, while others are specific to the underlying architecture and are only available on the respective system.

Within HiDALGO we rely on tools that are readily available to each user or can be easily installed by a user, to collect profiles and/or traces, which can then be analysed with various tools that visualize profiles and traces. Our benchmarking process uses both profiles and traces, the former collected through *sampling* and the latter collected through *tracing*. *Sampling* collects data from the program counter at specific time intervals [10] to check how much time was spent on a particular function [12][10]. *Tracing* collects performance data during a particular event such as a function call [10]. However, the user needs to specify which function to trace [12]. Tracing is usually a heavy process when the application is running for a long time on large number of cores. On the other hand, sampling is mostly used to get an initial overview of the work distribution [10].

To successfully achieve the project's objectives, HiDALGO tools must be carefully selected considering the following:

1. HiDALGO needs tools for profiling applications written in languages which are emerging in the HPC community, including Python.
2. Portability of benchmarking and profiling tools across various architectures as well as the usability of collected profiles and traces are of utmost importance.

Therefore, although we do make use of the pre-installed tools, we focus on tools that are widely-available, supported by multiple architectures and vendors, maintained and up-to-date, which provide information about the performance of applications that can be easily interpreted and re-used in the future not only by the project but also by the HPC and HPDA communities. We note that, although some tools are pre-installed on one or all HiDALGO systems, we may rely on different versions of the tools for the analysis performed in this deliverable.

Name	Hazelhen	Eagle	Scope
PAPI	Yes	No	Hardware counters measurement
Score-P	Yes	No	Profiling and trace recording
Intel Advisor	Yes	Yes	Design and optimization

Vampir	Yes	Yes	Trace visualization and analysis
Intel Inspector	Yes	Yes	Performance optimization
CrayPAT	Yes	No	Performance analysis
Intel VTune	Yes	Yes	Performance analysis
Extrae	Yes	No	Trace recording
Cube	Yes	No	Profile visualization
Paraver	Yes	Yes	Trace visualization

Table 2.1: Pre-installed performance analysis tools on Hazelhen and Eagle

2.2.2 Description of pre-installed application analysis tools

PAPI

PAPI aims to provide the tool designer and application engineer with a consistent interface and methodology for use of the performance counter hardware found in most major microprocessors. It enables software engineers to see the relation between software performance and processor events in near real time [6].

- Link: http://icl.cs.utk.edu/papi/index.html

Score-P

Score-P is a software system that provides a measurement infrastructure for profiling, event trace recording, and online analysis of HPC applications. Score-P supports multiple programming paradigms, i.e., MPI, SHMEM, OpenMP, Pthreads, CUDA, OpenCL and OpenAcc. The call-path profiles obtained with Score-P can be subsequently analysed with tools like CUBE and TAU, while the traces, which follow the OTF2 format, can be analysed with Vampir and Scalasca. In addition, Score-P supports recording hardware events, interfacing with common tools like PAPI and PERF. While other tools are also suitable for collecting application profiles and/or traces, e.g. Extrae, Score-P offers bindings for multiple languages, including Python, with additional plugins (https://github.com/score-p). For the purposes of this deliverable, we use Score-P v4.1, as it allows for Python bindings. Although local installation is simple, we will opt for global installation on HiDALGO infrastructure in the future.

- Link: http://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-4.1/html/ (instructions for using Score-P with C/C++ applications)
- Link: https://github.com/score-p/scorep_binding_python (instructions for using Score-P with Python applications)

Intel Advisor

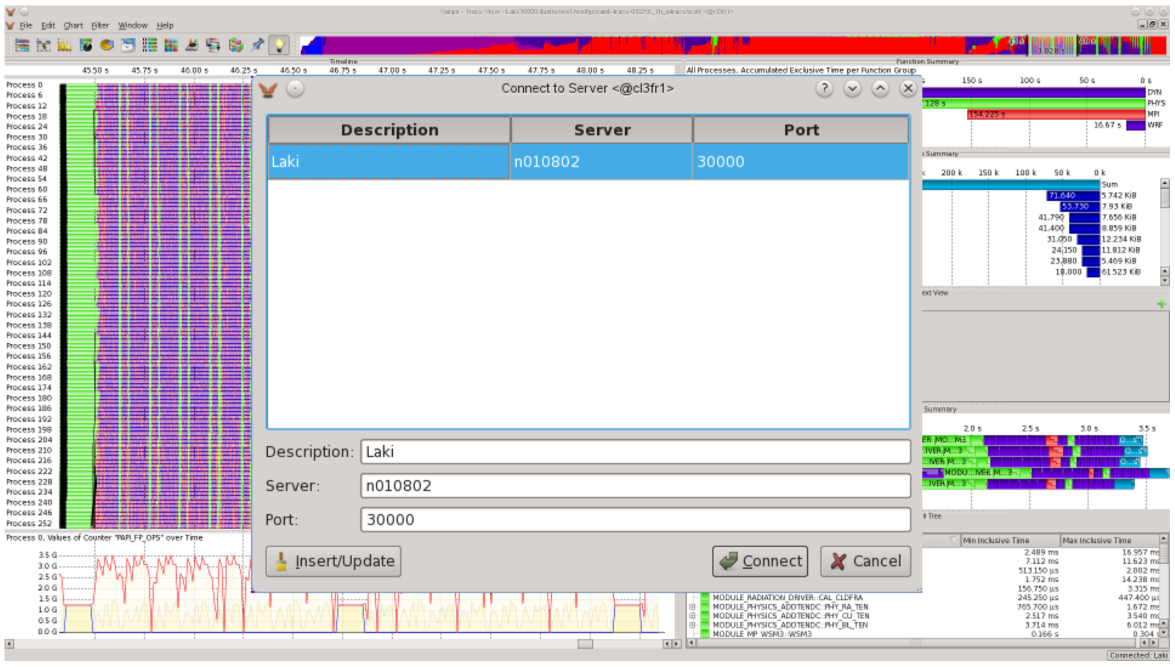
Intel Advisor is a threading assistant for C, C++, C# and Fortran. It guides developers through threading design, automating analyses required for fast and correct implementation. It helps developers to add parallelism to their existing C/C++ or Fortran programs [4].

➢ Link: https://software.intel.com/en-us/advisor

Vampir

The Vampir suite of tools offers scalable event analysis through a GUI which enables a fast and interactive rendering of very complex performance data. The suite consists of VampirTrace, Vampir and VampirServer. Ultra large data volumes can be analysed with a parallel version of VampirServer, loading and analysing the data on the compute nodes with the GUI of Vampir attached to it. Vampir is based on standard QT and works on desktop Unix workstations as well as on parallel production systems [8].

Vampir consists of a GUI interface and an analysis backend as shown in Figure 2.1. In order to use Vampir, you first need to generate a trace of the application, preferably using VampirTrace. The generated Open Trace Format (OTF) trace consists of a file for each MPI process (*.events.z), a trace definition file (*.def.z) and the master trace file (*.otf) describing the other files [8].


Connect to Server <@cl3fr1>

Description	Server	Port
Laki	n010802	30000

Description: Laki

Server: n010802

Port: 30000

Figure 2.1: Vampir GUI [8]

Using Vampire

To analyse small traces (< 500 MB of trace data), Vampir can be used standalone with the default backend:

```
vampire
```

Using VampireServer

For large-scale traces (> 500MB and up to many thousand MPI processes), the parallel VampirServer backend should be used (on compute nodes allocated through the queuing system). The user can then attach to it using vampir:

```
vampirserver start -n $((16*8 - 1))
```

On a new shell window, login to one of the login nodes of the system and open vampire:

```
vampire
```

Then, as shown in Figure 2.1, select "*Remote open*" and enter the host and port of the VampirServer. Proceed and select the trace you want to open.

- Link: https://vampir.eu/

Intel Inspector

Intel Inspector is able to determine errors related to threading and memory early in the development cycle to deliver reliable applications. It is a memory error and thread checker tool for C, C++, C# .NET, and Fortran developers designing serial and parallel applications [5]. It performs dynamic analysis detecting intermittent and non-deterministic errors on varying degrees of analysis. The tool detects memory leaks and problems and locates them. In terms of threading analysis, it detects and locates deadlocks and data races.

Intel Inspector has two interfaces available: graphical and command line. To use Intel Inspector, the code must be compiled with -g option to get debug information and preferably with -O0 to ensure the debug information accurately reflects the source code location.

- Link: https://software.intel.com/en-us/intel-inspector
- Tutorial: https://software.intel.com/en-us/inspector-tutorial-windows-memory-cplusplus

CrayPAT

CrayPAT, offered by Cray, is a performance analysis tool for XC platforms [1]. It supports Fortran, C, C++, UPC, OpenMP, Pthreads and SHMEM [12]. The CrayPAT tool set contains a simplified and easy-to-use version of Perftools-lite which provides automated performance analysis information [1]. CrayPAT supports two mechanisms to collect or measure performance data, sampling and tracing.

Using CrayPAT

To use CrayPAT for automatic performance analysis, first a .apa file needs to be generated using the commands below.

After building the application, the following command instruments the application:

```
pat_build –O apa a.out
```

The result of the above command will be an instrumented program file a.out+pat.

Now use the following command to run the application to get top time-consuming routines, using sampling:

```
aprun ... a.out+pat
```

Running the +pat binary creates a .xf data file or a directory containing multiple .xf files. To generate a single *.ap2 file, containing all performance data, use pat_report to combine information from *.xf output.

Intel VTune

Intel VTune Amplifier XE for Linux allows to analyse algorithm choices, identify serial and parallel code bottlenecks, and understand how to better utilize available hardware resources and thus speed up the application execution. Moreover, it reveals where the application is spending time, identifies the most time-consuming program units (hotspots), and detects hardware usage bottlenecks for a sample application. The profiler has a GUI interface which presents analysis findings in a synthetized way what simplifies interpretation.

- Link: https://software.intel.com/en-us/vtune

Extrae

The Extrae library allows MPI communication events of a parallel program to be recorded as a trace file [2]. Extrae can be configured through an XML file. To set up the environment for the run use the following command:

```
export LD_PRELOAD=$EXTRAE_HOME/lib/libmpitrace.so
export EXTRAE_CONFIG_FILE=extrae.xml
```

And finally run the application as usual [2].

- Link: https://tools.bsc.es/extrae

Cube

Cube is used as performance report explorer for Scalasca and Score-P. It is a generic tool for displaying a multi-dimensional performance space consisting of dimensions such as performance metric, call path, and system resource [11].

- Link: http://scalasca.org/software/cube-4.x/documentation.html

Paraver

Based on an easy-to-use Motif GUI, Paraver was developed to respond to the need to have a qualitative global perception of the application behaviour by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems. Paraver is able to perform concurrent comparative analysis of several traces and provides a large amount of information useful to improve the decisions on whether and where to invest the programming effort to optimize an application [7].

Customizable semantics of the visualized information facilitates extending the tool to support new performance data or new programming models, while sharing views of the trace file enables cooperative work. Another degree of freedom is the building of derived metrics which are not hardwired on the tool but programmed. The tool offers a large set of time functions, a filter module and a mechanism to combine two time lines.

- Link: https://tools.bsc.es/paraver
- GitHub: https://github.com/bsc-performance-tools/wxparaver

2.2.3 HiDALGO storage infrastructure and tools

In order to establish collaborative work on the project sources, HLRS provide access to the installation of the `FusionForge` at https://projects.hlrs.de. `FusionForge` is a web-based project management and collaboration environment, which includes services for project hosting, version control (`Subversion`, `Git`, etc.), code reviews, ticketing (issues, support), release management, continuous integration, and messaging.

In order to use personal `Fusion Forge` services, the user should login to https://projects.hlrs.de and select the tab `"My Page"`. In particular, this tab contains subtabs `"Personal page"`, `"Account"`, and `"Register Project"`. Subtab `"Register Project"` allows to request for new project repository. While registering the new repository, the user must specify project full name, project purpose and summary, project public description, and preferable source control manager (SCM). The user can select between `Git`, `Subversion`, `Mercurial`, or datastore without SCM. Subtab `"Account"` serves for managing user account (change password, set up language, timezone, theme, country, email address, etc.). Subtab `"Personal page"` usually contains general information about projects where the user participates in. Its appearance can be easily tuned by adding widgets and customizing layouts. From this subtab, the user can get access to the project pages.

Each project pages contains tabs `"Summary"`, `"Admin"` (if the user has admin permissions for the project), `"SCM"` and others. Tab `"SCM"` assembles information about project repository: path for cloning, aggregate data about repository history, a web interface for manipulating the repository (e.g., `gate`), etc. Tab `"Summary"` holds project-related widgets that can be selected by user. The default set of widgets includes project description, references to the latest file releases, quick access to public services of the project (Home Page, Project Files, Mailing Lists, Surveys, SCM repository), and list of project members. Users with administrative rights have access to `"Admin"` tab. This tab allows to update project information, update project users (add/remove users, change their roles, etc.), view basic project statistics (e.g., see screenshot Fig. 1 with the bar plot of commits history).

In the context of HiDALGO WP 3, we intend to create `hidalgo-wp3` project with `Git` repository that stores relevant information about code optimization and benchmarking. It can be further cloned with:

```
git clone https://scm.projects.hlrs.de/authscm/sgogolenko/git/hidalgo-wp3/hidalgo-wp3.git
```

Pushing new commits to this repository usually consists of the following steps:

```
git commit -m"[commit-tag] Commit short description"
git pull --rebase
git push
```

2.3 The HiDALGO benchmarking process

2.3.1 Repository structure

As it is important to track changes in the status of applications and correlate collected measurements with scalability results, HiDALGO has defined the following structure for the shared repository:

- Each application has its own directory in the `hidalgo-wp3` project repository (`/RefugeeMigration`, `/UrbanPollution`, `/SocialMedia`).
- For each application, a subdirectory is created for each different system (e.g. `/RefugeeMigration/Hazelhen_HLRS`, `/RefugeeMigration/Eagle_PSNC`)
- Separate directories are created to denote different configurations in terms of compiler versions, library versions, etc. For example, if an application is benchmarked with two different MPI versions on the same system, then two different directories are created. We use incremental numbering for different configurations (e.g. `/RefugeeMigration/Hazelhen_HLRS/conf00`, `/RefugeeMigration/Hazelhen_HLRS/conf01`).

  Each of these directories stores the following:

  - A `conf.yaml` file which effectively describes the system configuration. It lists the versions of used compilers and libraries, as well as any other configuration options, e.g. runtime parameters for a library.
  - A set of files listing the parameters together with their values used for the different executions of the application. The files are named `param_N.yaml`, where `N` denotes the N-th execution of the application.
  - Sets of measurements for the whole application or for one of its components using the CSV format (`.csv` files). The name of each CSV file follows a specific convention for each component of each application. Specifically, it must include the application name, the component name, the name of the xml file that contains the configuration parameters for the application and/or the specific component, and a timestamp.

    e.g. `AppName-AppComponent-param_N-YYMMDD_hhmm.csv`
  - Sets of profiling results for the whole application or one of its components. Each set is stored in a timestamped subdirectory using the naming convention `profiling-YYMMDD_hhmm`.

    Each of these subdirectories contains:

    - A file holding the configuration parameters used for the profiling of the specific component or the application, named `AppName-AppComponent-param.yaml`.

- One or more directories with profiles or traces for the application or the particular component, for a number of nodes, cores, and processes per node. The name of each one of these directories follows a specific convention. In particular, it starts with a prefix (`profile-` for profiles or `trace-` for traces), which is then followed by the execution configuration. For example, for a trace collected on `n` nodes, `c` cores per node, `p` MPI processes and `d` threads per process, the name of the directory would be: `trace-n_nodes-c_cores-p_procs-d_threads`.

Figure 2.2 illustrates the HiDALGO benchmarking repository structure, showcasing the directories and files stored for executing and profiling the Flee component of the Migration pilot at Hazelhen.

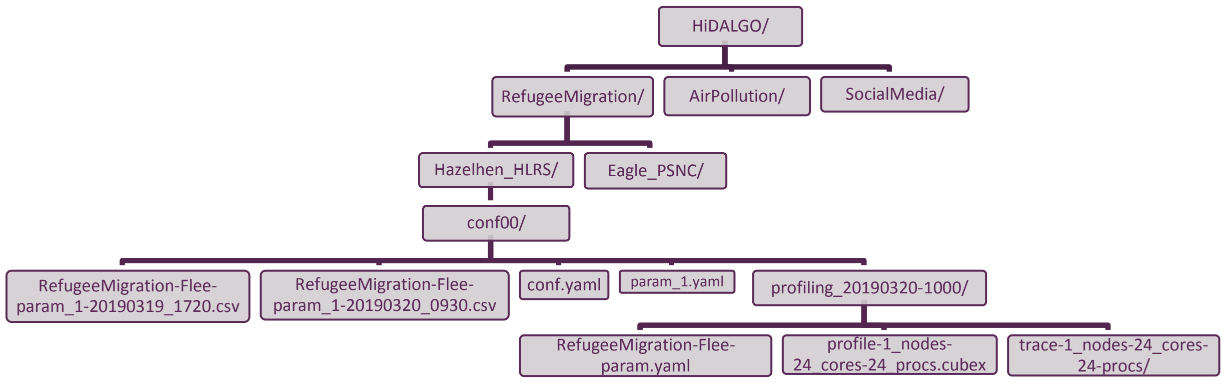

HiDALGO/
RefugeeMigration/
AirPollution/
SocialMedia/
Hazelhen_HLRS/
Eagle_PSNC/
conf00/
RefugeeMigration-Flee-param_1-20190319_1720.csv
RefugeeMigration-Flee-param_1-20190320_0930.csv
conf.yaml
param_1.yaml
profiling_20190320-1000/
RefugeeMigration-Flee-param.yaml
profile-1_nodes-24_cores-24_procs.cubex
trace-1_nodes-24_cores-24-procs/

Figure 2.2 HiDALGO benchmarking repository structure

2.3.2 Reporting guidelines

Each major software component of an application benchmarked and profiled by HiDALGO, is expected to be able to report its own timing results and any other metrics specific to the application, both for the component as a whole and for critical kernels or phases within the component (e.g. startup/initialisation phase, iterative phases, etc.). All the results (time and other metrics) are reported in a CSV file.

To facilitate the storing, tracking and analysis of results, HiDALGO has defined a specific structure that must be followed by each CSV results file. Specifically, each file is structured as follows:

- The first column(s) specify the problem size and/or instantiation parameters of the problem.
- The next three columns refer to the number of nodes (“`Nodes`”), number of cores per node (“`Cores/Node`”), number of MPI processes used (“`MPI_Procs`”). If the

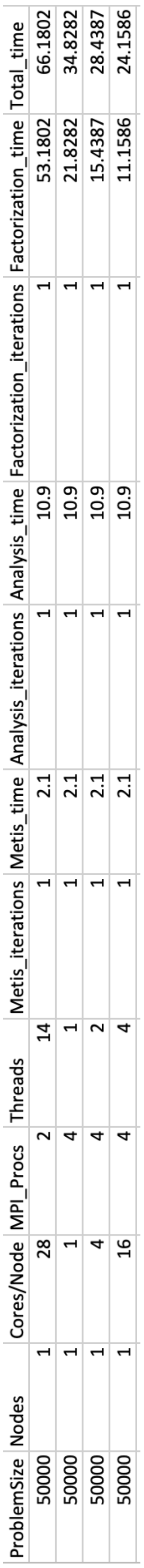
Document name:	D3.1 Report on Benchmarking and Optimisation					**Page:**	18 of 37
Reference:	D3.1	**Dissemination:**	PU	**Version:**	1.0	**Status:**	Final

component is parallelized using hybrid MPI+OpenMP, they are followed by a fourth column that refers to the number of threads per MPI process (“`Threads`”).

- A set of columns for each phase of the component we need to collect measurements for. The columns are named following a specific convention. In particular, their names start with a prefix, which is the name of the phase (custom to the application component), followed by the name of the metric. For example, for phase “PhaseX” of a component, the following columns are created: “`PhaseX_iterations`”, “`PhaseX_time`”, “`PhaseX_metric1`”, “`PhaseX_metric2`”, etc. These columns denote the number of iterations for the particular phase, its execution time, and any other metrics collected for this particular phase.
- The last columns include the total execution time of the component and aggregate metrics. These columns are named using the prefix “`Total_`”.

The proposed template for reporting is easily extensible with additional application phases and metrics. In case of variability in runtimes or other metrics between multiple runs, the template can be extended with a column denoting the number of runs (“`Runs`”), while, for each metric measurement, the mean observed value and its confidence interval is reported instead. For example, instead of a single column named “`PhaseX_time`”, we use multiple columns named “`PhaseX_time_mean`”, “`PhaseX_time_CI-cl`”, where `cl` is the confidence level for the confidence interval.

Figure 2.3 showcases the CSV file that stores the results for the Validation module of the Social Networks pilot. For a specific problem size, the application has been executed 4 times, each one using a different configuration, i.e. different nodes, cores, MPI processes and threads. The Validation module has 3 different phases, namely Metis, Analysis and Factorization. For each phase the CSV file holds its iterations and the time it took to be executed. Finally, the CSV file reports the total execution time of each run.



ProblemSize	Nodes	Cores/Node	MPI_Procs	Threads	Metis_iterations	Metis_time	Analysis_iterations	Analysis_time	Factorization_iterations	Factorization_time	Total_time
50000	1	28	2	14	1	2.1	1	10.9	1	53.1802	66.1802
50000	1	1	4	1	1	2.1	1	10.9	1	21.8282	34.8282
50000	1	4	4	2	1	2.1	1	10.9	1	15.4387	28.4387
50000	1	16	4	4	1	2.1	1	10.9	1	11.1586	24.1586

Figure 2.3 : Example of a CSV file

2.3.3 Profiling guidelines

In order to define a uniform methodology for profiling applications in the context of HiDALGO, we rely on Score-P.

More specifically, to profile HPC components of applications, we use Score-P to collect either application profiles or application traces. The collected results are stored in the repository as described in Sections 2.3.1 and 2.3.2. Score-P profiles and traces can then be analysed with different tools. More importantly, Score-P allows us to collect any required information of an application in the same format, which can later be re-used or compared across systems and/or applications. Score-P collects timing measurements and call trees for parallel applications at profiling mode, while it captures all application events when in tracing mode.

Score-P primarily targets MPI. It can profile or record events for a full application, but also allows for instrumentation and measurement of specific parts of the code. Its interface with PAPI and/or perf allows also for collection of performance counters measurements for a specific phase of an application. Finally, it also allows for memory recording and I/O recording.

3 Initial Benchmarking Findings

3.1 HiDALGO HPC/HPDA infrastructure

As a reference point for the HiDALGO HPC infrastructure (Hazelhen and Eagle), Deliverable D3.1 reports the benchmarking results for the HPL and HPCG benchmarks, both on full systems and on single nodes.

These two benchmarks act as indicators of diverse performance properties. The HPL benchmark, having O(n) operational intensity, indicates the peak performance of a system in heavily computationally-intensive applications. On the other hand, the HPCG benchmark, having O(1) operational intensity, namely a very low flop-per-byte ration, indicates the performance of a system when the memory subsystem is stressed. In addition, the HPCG benchmark exhibits complex communication patterns and thus is capable of exposing performance bugs in common communication operations at large scale.

Regarding ECMWF infrastructure, which as explained in Section 2.1.3 has a different operation than other two supercomputing centres, Deliverable D3.1 reports I/O performance results specific to the high-resolution weather forecast (HRES) and the ensemble of 15-day forecasts (ENS).

3.1.1 USTUTT Benchmarking results

Table 3.1 presents HPL and HPCG reference results on Hazelhen, on single node and on the full system. HPL results show the peak node and system performance, while HPCG results show the system performance under a memory-bound workload with significantly higher communication overheads. On a single node, HPCG achieves about 17% of the HPL performance, and about 14% in parallel efficiency on the full system.

Benchmark	Problem size	Nodes	Cores/Node	MPI processes/ Node	Memory/ Node	Performance
HPL	20000	1	24 (2x12)	1	128GB	777.233 GFlop/s
HPCG	Global: 192x192x192 Local: 192x192x192	1	24 (2x12)	1	128GB	131.322GFlop/s
HPL	4973760	7712	24 (2x12)	2	128GB	5640.2TFlop/s
HPCG	Global: 3072 x 5952 x 5952 Local: 192 x 192 x 192	7688	24 (2x12)	2	128GB	138 TFlop/s

Table 3.1: HPL and HPCG reference performance results for Hazelhen (HLRS)

3.1.2 PSNC Benchmarking results

Table 3.2 presents HPL and HPCG reference results on Eagle, on a single node and on the full system. For Eagle, HPL performance is slightly higher on a single node compared to Hazelhen, due to the 4 additional cores per node. HPCG achieves about 5% of the HPL performance, indicating that a node on Eagle is more sensitive to the performance of memory-bound workloads compared to a node in Hazelhen. The exact problem size for HPCG could not be verified for Eagle, so we cannot make any assumptions about the scalability of the interconnection network at this time.

Benchmark	Problem size	Nodes	Cores/Node	MPI processes/ Node	Memory/ Node	Performance
HPL	120960	1	28 (2x24)	28	128GB	894.7 GFlop/s
HPCG	Local: 16x16x16	1	28 (2x24)	28	128GB	44.4102GFlop/s
HPL	2739840	1233	32984	2	64GB/128GB/256GB	1013.72TFlop/s
HPCG	unknown	1233	32984	2	64GB/128GB/256GB	8.53 TFLop/s

Table 3.2: HPL and HPCG reference performance results for Eagle (PSNC)

3.1.3 ECMWF Benchmarking results

The ECMWF model produces raw model output for *global fields* in the spectral space (spherical harmonic fields) and physical space (reduced Gaussian grid, reduced lat/lon). These fields still need further post-processing to create *user-specific* weather products. Member states and clients very often require specific tailored products, e.g. temperature in the whole country of Hungary or precipitation on a coarse lat-lon grid. Product Generation application applies users' requirements to raw model output to get user data fields, called *products.*

summarizes the reference performance results from production runs on the ECMWF infrastructure. The HRES column refers to the high-resolution forecast and the ENS column refers to the ensemble of forecasts.

Metrics	HRES	ENS
Total		
Number of fields read	1,532,803	24,552,389
Number of products written	22,875,147	114,909,155
Writing volume	14.11 TB	11.13 TB
Writing rate	648.6 MB/s	393.4 MB/s
Main model run		
Number of fields read	488,336	9,432,089
Number of products written	11,614,366	65,343,476
Writing volume	9.05 TB	9.15 TB
Writing rate	625.0 MB/s	506.0 MB/s
Boundary-conditions run		
Number of fields read	1,044,467	15,120,300
Number of products written	11,260,781	49,565,679
Writing volume	5.06 TB	1.98 TB
Writing rate	672.2 MB/s	280.7 MB/s

Table 3.3: Reference performance results for ECMWF infrastructure

3.2 HiDALGO Pilots

3.2.1 Migration Pilot

3.2.1.1 Current status of the workflow

A detailed description of the Migration pilot workflow can be found in Deliverable 4.1. Its core component is the Flee simulation framework. Flee is an agent-based modelling framework that simulates the movement of individuals across geographical locations. The code is written in Python and parallelized with MPI, using the py4mpi module [9]. A public release of the code is available on github[1].

For this deliverable, we benchmark the parallel version of the code using sample input data, which contain 2 conflict zones, 2 towns and 3 camps. The simulation starts with a configurable number of initial agents/refugees per campus. As the simulation progresses in time (also

[1] https://github.com/djgroen/flee-release

configurable number of time steps), at every time step, new conflict zones and closures are introduced and refugee movements are simulated accordingly on the network graph. In addition, the test inserts an additional 1000 agents per time step. This particular execution scenario is not I/O-intensive, as conflict and refugee data is loaded in advance of the main simulation.

3.2.1.2 Preliminary benchmarking findings

We present scalability results for the parallel test on both Hazelhen and Eagle, for up to 4 compute nodes, scaling up the number of MPI processes. We use 120-time steps for the simulation and we run tests with 300K agents (100K agents per campus) and 900K agents (300K agents per campus). The execution time is measured using the `GNU time` command and refers to the full simulation, including starting up the MPI processes and the initialization of the agents. We did not scale the application on more nodes, since our preliminary results, shown below, already result in low execution times for the selected parallel test and suffice to expose the scalability behaviour of the application.

Figure 3.1 (both axes in logarithmic scale) and Figure 3.2 show the results of scaling Flee on Hazelhen for up to 4 nodes (96 cores). Flee scales up well within a single node (up to 24 cores) for both problem sizes; however, its parallel efficiency is reduced when scaled to more nodes. In particular, the achieved efficiency is 64% for 24 cores (single node) for the larger problem size (900K agents), while being only 39% for 96 cores (4 nodes). For the smaller problem size, efficiency is consistently lower across all core counts.

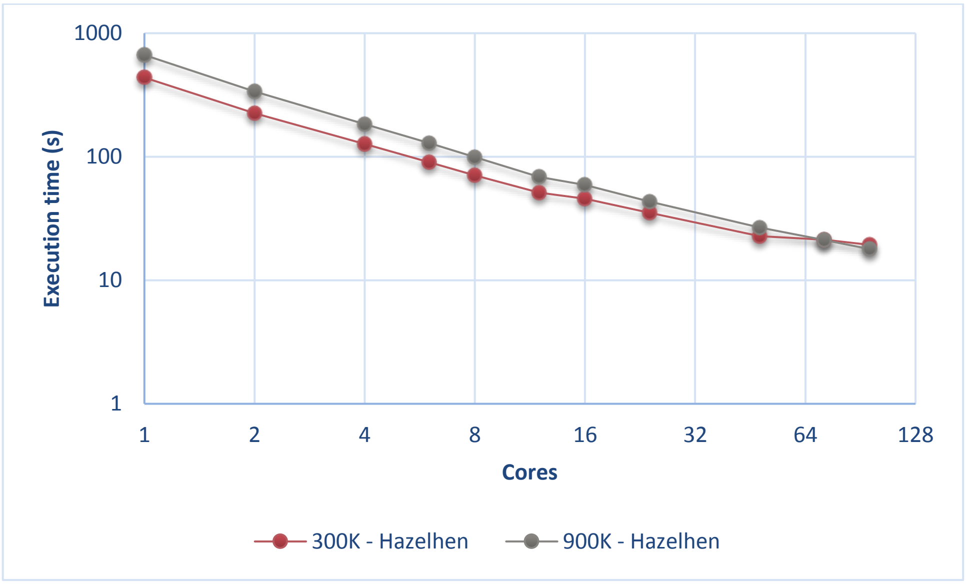


Figure 3.1: Scaling Flee on Hazelhen (HLRS) – Execution time

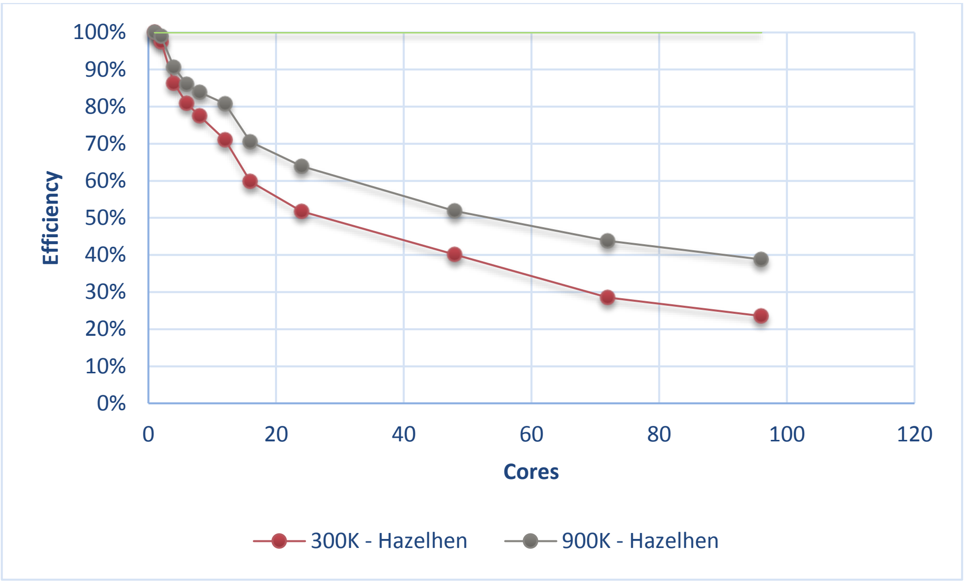


Figure 3.2: Scaling Flee on Hazelhen (HLRS) – Efficiency

Similar behaviour is observed on Eagle, as illustrated by Figure 3.3 (both axes in logarithmic scale) and Figure 3.4**¡Error! No se encuentra el origen de la referencia.**. For a single node (28 cores), parallel efficiency for the larger problem size is 67%, while dropping to 28% for 4 nodes (112 cores). What is noteworthy is that on Eagle, parallel efficiency is almost equal for the two different problem sizes for up to 24 cores. In addition, on Hazelhen, there appears to be an increasing difference in parallel efficiency between the two problem sizes, while on Eagle, the difference seems to remain almost constant as the core count increases. Those differences can be further investigated through profiling of the application.

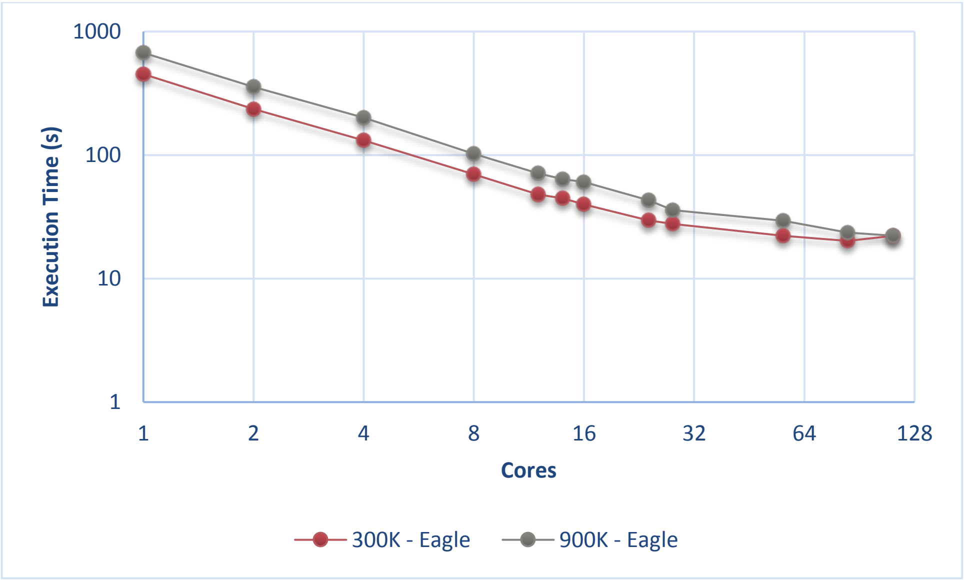


Figure 3.3: Scaling Flee on Eagle (PSNC) - Execution time

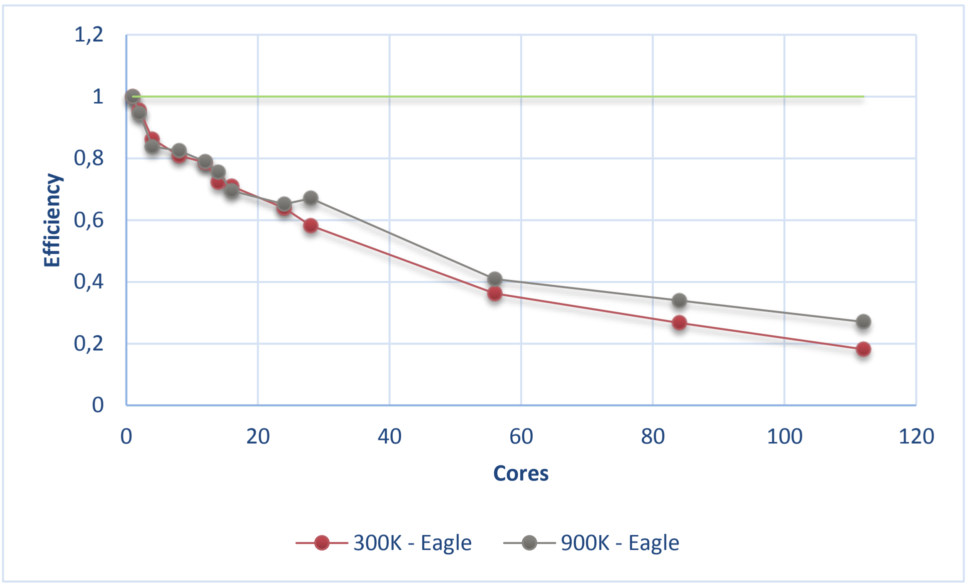
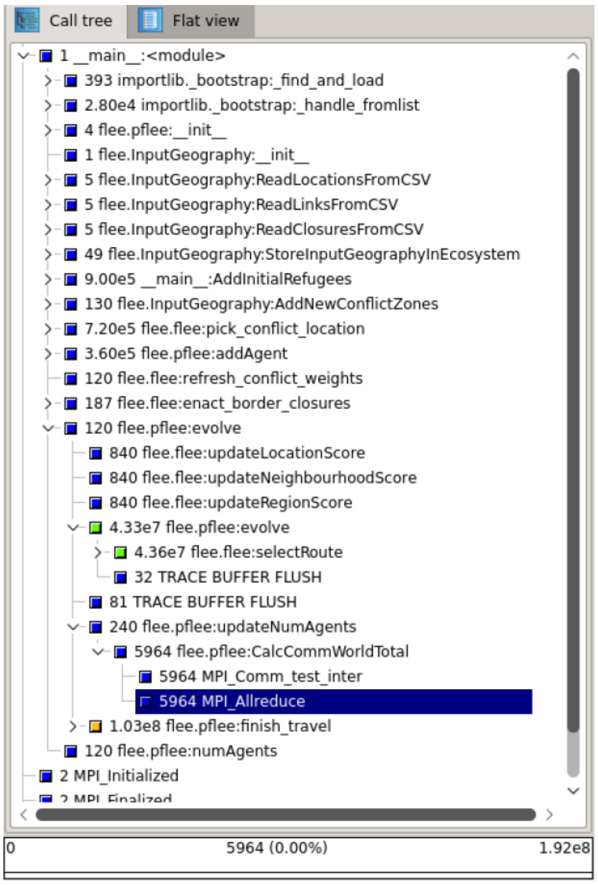


Figure 3.4: Scaling Flee on Eagle (PSNC) - Efficiency

3.2.1.3 Profiling findings

In order to further understand the scalability problems of Flee, we have profiled the parallel test for 300K agents on both systems, using Score-P to collect both a profile and a trace of the application. We use Cube to visualise and analyse application profiles and Vampir to visualise and analyse the OTF2 traces. We have collected traces for 1 process, half a node (12 cores on Hazelhen, 14 cores on Eagle), full node (24 cores on Hazelhen, 28 cores on Eagle) and 2 nodes.

By analysing the profile for a single process on Hazelhen using Cube (Figure 3.5), we have identified that Flee in this execution scenario calls the `MPI_Allreduce` functions an excessive amount of times, i.e. almost 6000 times. This is actually due to the parallelization of the application: a call to `MPI_Allreduce` is performed at every time step, for each location and link in the simulation, in order for all processes to have a global view of the number of simulated agents at any time. As for larger location graphs, this may result in an excessive number of calls, a more efficient rearrangement of data, combined with merging these calls, could result in improved performance.

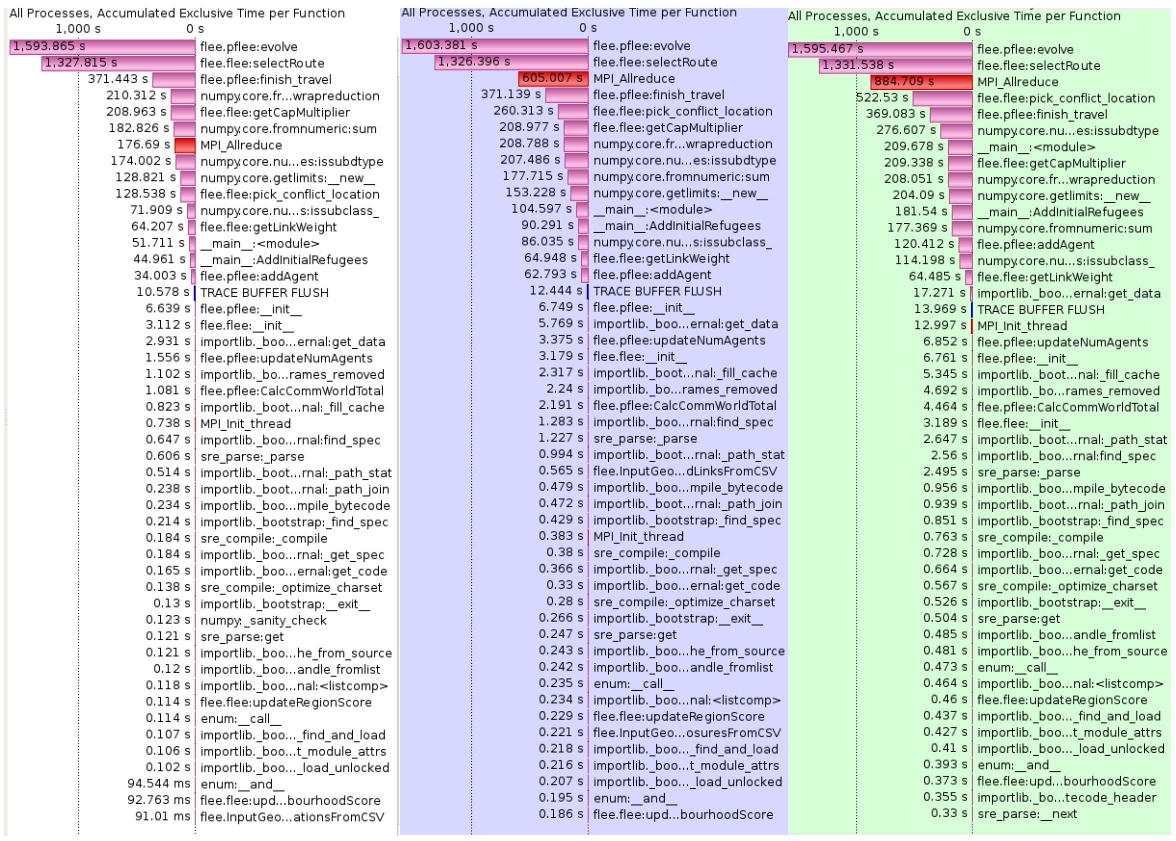


```
Call tree    Flat view
v 1 __main__:<module>
  > 393 importlib._bootstrap:_find_and_load
  > 2.80e4 importlib._bootstrap:_handle_fromlist
  > 4 flee.pflee:__init__
    1 flee.InputGeography:__init__
  > 5 flee.InputGeography:ReadLocationsFromCSV
  > 5 flee.InputGeography:ReadLinksFromCSV
  > 5 flee.InputGeography:ReadClosuresFromCSV
  > 49 flee.InputGeography:StoreInputGeographyInEcosystem
  > 9.00e5 __main__:AddInitialRefugees
  > 130 flee.InputGeography:AddNewConflictZones
  > 7.20e5 flee.flee:pick_conflict_location
  > 3.60e5 flee.pflee:addAgent
    120 flee.flee:refresh_conflict_weights
  > 187 flee.flee:enact_border_closures
  v 120 flee.pflee:evolve
      840 flee.flee:updateLocationScore
      840 flee.flee:updateNeighbourhoodScore
      840 flee.flee:updateRegionScore
    v 4.33e7 flee.pflee:evolve
      > 4.36e7 flee.flee:selectRoute
        32 TRACE BUFFER FLUSH
      81 TRACE BUFFER FLUSH
    v 240 flee.pflee:updateNumAgents
      v 5964 flee.pflee:CalcCommWorldTotal
          5964 MPI_Comm_test_inter
          5964 MPI_Allreduce
    > 1.03e8 flee.pflee:finish_travel
    120 flee.pflee:numAgents
  2 MPI_Initialized
  2 MPI_Finalized
0                    5964 (0.00%)                    1.92e8
```

Figure 3.5: Call tree (number of calls) of Flee with 1 MPI process on Hazelhen using Cube.

As `MPI_Allreduce` is an intensive all-to-all communication function, we expect it to be a major bottleneck for scaling Flee on high numbers of nodes. A further analysis of the traces of the application on Hazelhen using Vampir is shown in Figure 3.6, which visualizes the accumulated exclusive time per function. The traces show that `MPI_Allreduce` becomes the third most time-consuming function when already filling a single node on Hazelhen.

The two most time-consuming functions, namely `flee:pflee:evolve` and `flee:flee:selectRoute` scale linearly up to 48 cores (the accumulated exclusive time remains constant). On the contrary, the time consumed in the `MPI_Allreduce` function grows significantly even within a single node, thus impeding the scalability of the application. Another interesting finding is that the function `flee:flee:pick_conflict_location` does not scale linearly, becoming the fourth most time-consuming function when utilizing 48 cores.



All Processes, Accumulated Exclusive Time per Function (12 cores)

1,000 s 0 s

1,593.865 s	flee.pflee:evolve
1,327.815 s	flee.flee:selectRoute
371.443 s	flee.pflee:finish_travel
210.312 s	numpy.core.fr...wrapreduction
208.963 s	flee.flee:getCapMultiplier
182.826 s	numpy.core.fromnumeric:sum
176.69 s	MPI_Allreduce
174.002 s	numpy.core.nu...es:issubdtype
128.821 s	numpy.core.getlimits:__new__
128.538 s	flee.flee:pick_conflict_location
71.909 s	numpy.core.nu...s:issubclass_
64.207 s	flee.flee:getLinkWeight
51.711 s	__main__:<module>
44.961 s	__main__:AddInitialRefugees
34.003 s	flee.pflee:addAgent
10.578 s	TRACE BUFFER FLUSH
6.639 s	flee.pflee:__init__
3.112 s	flee.flee:__init__
2.931 s	importlib._boo...ernal:get_data
1.556 s	flee.pflee:updateNumAgents
1.102 s	importlib._bo...rames_removed
1.081 s	flee.pflee:CalcCommWorldTotal
0.823 s	importlib._boot...nal:_fill_cache
0.738 s	MPI_Init_thread
0.647 s	importlib._boo...rnal:find_spec
0.606 s	sre_parse:_parse
0.514 s	importlib._boot...rnal:_path_stat
0.238 s	importlib._boot...rnal:_path_join
0.234 s	importlib._boo...mpile_bytecode
0.214 s	importlib._bootstrap:_find_spec
0.184 s	sre_compile:_compile
0.184 s	importlib._boo...rnal:_get_spec
0.165 s	importlib._boo...ernal:get_code
0.138 s	sre_compile:_optimize_charset
0.13 s	importlib._bootstrap:__exit__
0.123 s	numpy:_sanity_check
0.121 s	sre_parse:get
0.121 s	importlib._boo...he_from_source
0.12 s	importlib._boo...andle_fromlist
0.118 s	importlib._boo...nal:<listcomp>
0.114 s	flee.flee:updateRegionScore
0.114 s	enum:__call__
0.107 s	importlib._boo..._find_and_load
0.106 s	importlib._boo...t_module_attrs
0.102 s	importlib._boo..._load_unlocked
94.544 ms	enum:__and__
92.763 ms	flee.flee:upd...bourhoodScore
91.01 ms	flee.InputGeo...ationsFromCSV

All Processes, Accumulated Exclusive Time per Function (24 cores)

1,000 s 0 s

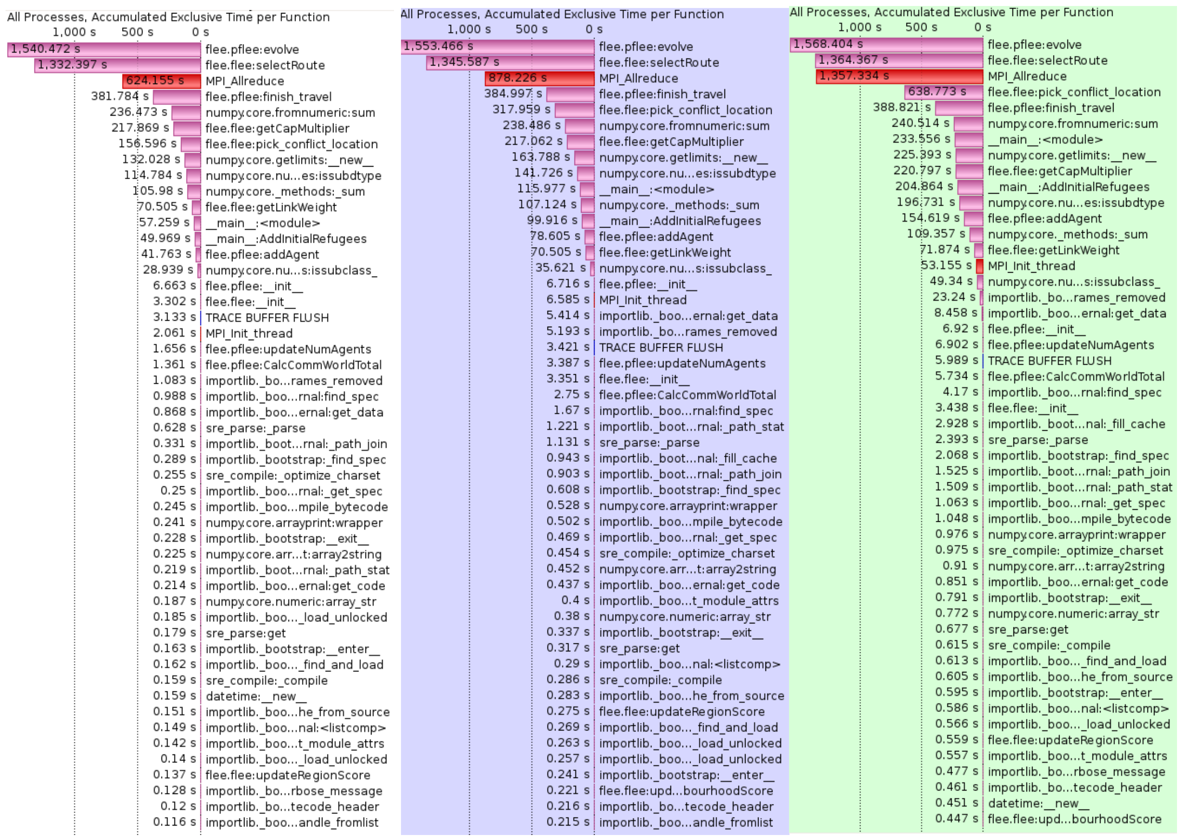
1,603.381 s	flee.pflee:evolve
1,326.396 s	flee.flee:selectRoute
605.007 s	MPI_Allreduce
371.139 s	flee.pflee:finish_travel
260.313 s	flee.flee:pick_conflict_location
208.977 s	flee.flee:getCapMultiplier
208.788 s	numpy.core.fr...wrapreduction
207.486 s	numpy.core.nu...es:issubdtype
177.715 s	numpy.core.fromnumeric:sum
153.228 s	numpy.core.getlimits:__new__
104.597 s	__main__:<module>
90.291 s	__main__:AddInitialRefugees
86.035 s	numpy.core.nu...s:issubclass_
64.948 s	flee.flee:getLinkWeight
62.793 s	flee.pflee:addAgent
12.444 s	TRACE BUFFER FLUSH
6.749 s	flee.pflee:__init__
5.769 s	importlib._boo...ernal:get_data
3.375 s	flee.pflee:updateNumAgents
3.179 s	flee.flee:__init__
2.317 s	importlib._boot...nal:_fill_cache
2.24 s	importlib._bo...rames_removed
2.191 s	flee.pflee:CalcCommWorldTotal
1.283 s	importlib._boo...rnal:find_spec
1.227 s	sre_parse:_parse
0.994 s	importlib._boot...rnal:_path_stat
0.565 s	flee.InputGeo...dLinksFromCSV
0.479 s	importlib._boo...mpile_bytecode
0.472 s	importlib._boot...rnal:_path_join
0.429 s	importlib._bootstrap:_find_spec
0.383 s	MPI_Init_thread
0.38 s	sre_compile:_compile
0.366 s	importlib._boo...rnal:_get_spec
0.33 s	importlib._boo...ernal:get_code
0.28 s	sre_compile:_optimize_charset
0.266 s	importlib._bootstrap:__exit__
0.247 s	sre_parse:get
0.243 s	importlib._boo...he_from_source
0.242 s	importlib._boo...andle_fromlist
0.235 s	enum:__call__
0.234 s	importlib._boo...nal:<listcomp>
0.229 s	flee.flee:updateRegionScore
0.221 s	flee.InputGeo...osuresFromCSV
0.218 s	importlib._boo..._find_and_load
0.216 s	importlib._boo...t_module_attrs
0.207 s	importlib._boo..._load_unlocked
0.195 s	enum:__and__
0.186 s	flee.flee:upd...bourhoodScore

All Processes, Accumulated Exclusive Time per Function (48 cores)

1,000 s 0 s

1,595.467 s	flee.pflee:evolve
1,331.538 s	flee.flee:selectRoute
884.709 s	MPI_Allreduce
522.53 s	flee.flee:pick_conflict_location
369.083 s	flee.pflee:finish_travel
276.607 s	numpy.core.nu...es:issubdtype
209.678 s	__main__:<module>
209.338 s	flee.flee:getCapMultiplier
208.051 s	numpy.core.fr...wrapreduction
204.09 s	numpy.core.getlimits:__new__
181.54 s	__main__:AddInitialRefugees
177.369 s	numpy.core.fromnumeric:sum
120.412 s	flee.pflee:addAgent
114.198 s	numpy.core.nu...s:issubclass_
64.485 s	flee.flee:getLinkWeight
17.271 s	importlib._boo...ernal:get_data
13.969 s	TRACE BUFFER FLUSH
12.997 s	MPI_Init_thread
6.852 s	flee.pflee:updateNumAgents
6.761 s	flee.pflee:__init__
5.345 s	importlib._boot...nal:_fill_cache
4.692 s	importlib._bo...rames_removed
4.464 s	flee.pflee:CalcCommWorldTotal
3.189 s	flee.flee:__init__
2.647 s	importlib._boot...rnal:_path_stat
2.56 s	importlib._boo...rnal:find_spec
2.495 s	sre_parse:_parse
0.956 s	importlib._boo...mpile_bytecode
0.939 s	importlib._boot...rnal:_path_join
0.851 s	importlib._bootstrap:_find_spec
0.763 s	sre_compile:_compile
0.728 s	importlib._boo...rnal:_get_spec
0.664 s	importlib._boo...ernal:get_code
0.567 s	sre_compile:_optimize_charset
0.526 s	importlib._bootstrap:__exit__
0.504 s	sre_parse:get
0.485 s	importlib._boo...andle_fromlist
0.481 s	importlib._boo...he_from_source
0.473 s	enum:__call__
0.464 s	importlib._boo...nal:<listcomp>
0.46 s	flee.flee:updateRegionScore
0.437 s	importlib._boo..._find_and_load
0.427 s	importlib._boo...t_module_attrs
0.41 s	importlib._boo..._load_unlocked
0.393 s	enum:__and__
0.373 s	flee.flee:upd...bourhoodScore
0.355 s	importlib._bo...tecode_header
0.33 s	sre_parse:__next

Figure 3.6: Flee trace analysis with Vampir on Hazelhen, for 12, 24, 48 cores (left to right).



All Processes, Accumulated Exclusive Time per Function (14 cores)

1,000 s 500 s 0 s

1,540.472 s	flee.pflee:evolve
1,332.397 s	flee.flee:selectRoute
624.155 s	MPI_Allreduce
381.784 s	flee.pflee:finish_travel
236.473 s	numpy.core.fromnumeric:sum
217.869 s	flee.flee:getCapMultiplier
156.596 s	flee.flee:pick_conflict_location
132.028 s	numpy.core.getlimits:__new__
114.784 s	numpy.core.nu...es:issubdtype
105.98 s	numpy.core._methods:_sum
70.505 s	flee.flee:getLinkWeight
57.259 s	__main__:<module>
49.969 s	__main__:AddInitialRefugees
41.763 s	flee.pflee:addAgent
28.939 s	numpy.core.nu...s:issubclass_
6.663 s	flee.pflee:__init__
3.302 s	flee.flee:__init__
3.133 s	TRACE BUFFER FLUSH
2.061 s	MPI_Init_thread
1.656 s	flee.pflee:updateNumAgents
1.361 s	flee.pflee:CalcCommWorldTotal
1.083 s	importlib._bo...rames_removed
0.988 s	importlib._boo...rnal:find_spec
0.868 s	importlib._boo...ernal:get_data
0.628 s	sre_parse:_parse
0.331 s	importlib._boot...rnal:_path_join
0.289 s	importlib._bootstrap:_find_spec
0.255 s	sre_compile:_optimize_charset
0.25 s	importlib._boo...rnal:_get_spec
0.245 s	importlib._boo...mpile_bytecode
0.241 s	numpy.core.arrayprint:wrapper
0.228 s	importlib._bootstrap:__exit__
0.225 s	numpy.core.arr...t:array2string
0.219 s	importlib._boot...rnal:_path_stat
0.214 s	importlib._boo...ernal:get_code
0.187 s	numpy.core.numeric:array_str
0.185 s	importlib._boo..._load_unlocked
0.179 s	sre_parse:get
0.163 s	importlib._bootstrap:__enter__
0.162 s	importlib._boo..._find_and_load
0.159 s	sre_compile:_compile
0.159 s	datetime:__new__
0.151 s	importlib._boo...he_from_source
0.149 s	importlib._boo...nal:<listcomp>
0.142 s	importlib._boo...t_module_attrs
0.14 s	importlib._boo..._load_unlocked
0.137 s	flee.flee:updateRegionScore
0.128 s	importlib._bo...rbose_message
0.12 s	importlib._bo...tecode_header
0.116 s	importlib._boo...andle_fromlist

All Processes, Accumulated Exclusive Time per Function (28 cores)

1,000 s 500 s 0 s

1,553.466 s	flee.pflee:evolve
1,345.587 s	flee.flee:selectRoute
878.226 s	MPI_Allreduce
384.997 s	flee.pflee:finish_travel
317.959 s	flee.flee:pick_conflict_location
238.486 s	numpy.core.fromnumeric:sum
217.062 s	flee.flee:getCapMultiplier
163.788 s	numpy.core.getlimits:__new__
141.726 s	numpy.core.nu...es:issubdtype
115.977 s	__main__:<module>
107.124 s	numpy.core._methods:_sum
99.916 s	__main__:AddInitialRefugees
78.605 s	flee.pflee:addAgent
70.505 s	flee.flee:getLinkWeight
35.621 s	numpy.core.nu...s:issubclass_
6.716 s	flee.pflee:__init__
6.585 s	MPI_Init_thread
5.414 s	importlib._boo...ernal:get_data
5.193 s	importlib._bo...rames_removed
3.421 s	TRACE BUFFER FLUSH
3.387 s	flee.pflee:updateNumAgents
3.351 s	flee.flee:__init__
2.75 s	flee.pflee:CalcCommWorldTotal
1.67 s	importlib._boo...rnal:find_spec
1.221 s	importlib._boot...rnal:_path_stat
1.131 s	sre_parse:_parse
0.943 s	importlib._boot...nal:_fill_cache
0.903 s	importlib._boot...rnal:_path_join
0.608 s	importlib._bootstrap:_find_spec
0.528 s	numpy.core.arrayprint:wrapper
0.502 s	importlib._boo...mpile_bytecode
0.469 s	importlib._boo...rnal:_get_spec
0.454 s	sre_compile:_optimize_charset
0.452 s	numpy.core.arr...t:array2string
0.437 s	importlib._boo...ernal:get_code
0.4 s	importlib._boo...t_module_attrs
0.38 s	numpy.core.numeric:array_str
0.337 s	importlib._bootstrap:__exit__
0.317 s	sre_parse:get
0.29 s	importlib._boo...nal:<listcomp>
0.286 s	sre_compile:_compile
0.283 s	importlib._boo...he_from_source
0.275 s	flee.flee:updateRegionScore
0.269 s	importlib._boo..._find_and_load
0.263 s	importlib._boo..._load_unlocked
0.257 s	importlib._boo..._load_unlocked
0.241 s	importlib._bootstrap:__enter__
0.221 s	flee.flee:upd...bourhoodScore
0.216 s	importlib._bo...tecode_header
0.215 s	importlib._boo...andle_fromlist

All Processes, Accumulated Exclusive Time per Function (56 cores)

1,000 s 500 s 0 s

1,568.404 s	flee.pflee:evolve
1,364.367 s	flee.flee:selectRoute
1,357.334 s	MPI_Allreduce
638.773 s	flee.flee:pick_conflict_location
388.821 s	flee.pflee:finish_travel
240.514 s	numpy.core.fromnumeric:sum
233.556 s	__main__:<module>
225.393 s	numpy.core.getlimits:__new__
220.797 s	flee.flee:getCapMultiplier
204.864 s	__main__:AddInitialRefugees
196.731 s	numpy.core.nu...es:issubdtype
154.619 s	flee.pflee:addAgent
109.357 s	numpy.core._methods:_sum
71.874 s	flee.flee:getLinkWeight
53.155 s	MPI_Init_thread
49.34 s	numpy.core.nu...s:issubclass_
23.24 s	importlib._bo...rames_removed
8.458 s	importlib._boo...ernal:get_data
6.92 s	flee.pflee:__init__
6.902 s	flee.pflee:updateNumAgents
5.989 s	TRACE BUFFER FLUSH
5.734 s	flee.pflee:CalcCommWorldTotal
4.17 s	importlib._boo...rnal:find_spec
3.438 s	flee.flee:__init__
2.928 s	importlib._boot...nal:_fill_cache
2.393 s	sre_parse:_parse
2.068 s	importlib._bootstrap:_find_spec
1.525 s	importlib._boot...rnal:_path_join
1.509 s	importlib._boot...rnal:_path_stat
1.063 s	importlib._boo...rnal:_get_spec
1.048 s	importlib._boo...mpile_bytecode
0.976 s	numpy.core.arrayprint:wrapper
0.975 s	sre_compile:_optimize_charset
0.91 s	numpy.core.arr...t:array2string
0.851 s	importlib._boo...ernal:get_code
0.791 s	importlib._bootstrap:__exit__
0.772 s	numpy.core.numeric:array_str
0.677 s	sre_parse:get
0.615 s	sre_compile:_compile
0.613 s	importlib._boo..._find_and_load
0.605 s	importlib._boo...he_from_source
0.595 s	importlib._bootstrap:__enter__
0.586 s	importlib._boo...nal:<listcomp>
0.566 s	importlib._boo..._load_unlocked
0.559 s	flee.flee:updateRegionScore
0.557 s	importlib._boo...t_module_attrs
0.477 s	importlib._bo...rbose_message
0.461 s	importlib._bo...tecode_header
0.451 s	datetime:__new__
0.447 s	flee.flee:upd...bourhoodScore

Figure 3.7: Flee trace analysis with Vampir on Eagle, for 14, 28, 56 cores (left to right).

Our analysis of traces on Eagle reveals a similar behaviour. Regarding the computational parts, this is expected, as the two systems utilise similar types of compute nodes. However, both the interconnection network and the MPI version used are different, affecting the communication parts of the application. As shown in Figure 3.7, `MPI_Allreduce` consumes a significant amount of time even for 14 processes on a single node of Eagle, and it does not scale at all outside of a single node, as indicated by the increase in time from 28 to 56 cores. This is an important point of comparison between the two systems: on Hazelhen, using the Cray-MPICH implementation of MPI, `MPI_Allreduce` does not scale well within a single node, i.e. its implementation for shared memory is not efficient, but it appears to scale on more nodes better than on Eagle, where we use OpenMPI v1.10.2. We intend to investigate the scalability of this particular function further in future steps of the benchmarking task.

Through the examination of call times of functions in our profiles, we also notice that the function `flee:flee:pick_conflict_location`, which is non-scalable beyond a single node, calls the numpy function `numpy.random.choice`. When simulating 300K agents, this function is called 1000 times at each time step by every MPI process. While the function itself does not consume much of the execution time, in practice, this is an inherent problem of the way Flee is currently parallelized; Flee is parallelized across agents but not along locations, which requires replicated computations across its processes for location update functions. This overhead grows proportionally with the size of the location graph, and therefore may prove to be a more significant bottleneck than our results currently show, as they have been obtained with a limited size location graph. Flee parallelization will be re-visited in the next steps of the project, so we expect this possible scalability bottleneck to be addressed.

Finally, regarding the scalability of different problem sizes, examination of call times of functions reveals fewer calls to the function `flee:pflee:evolve` for the larger problem size (900K agents), which is the reason for the equal execution times for the two problem sizes on 4 nodes.

3.2.2 Urban Pollution Pilot

3.2.2.1 Current status of the workflow

The core computational module of the Urban Pollution Pilot in its current status is the 3DAirQualityPrediction component, which performs the multicomponent CFD simulation of air flows in cities. A detailed description of the workflow can be found in Deliverable 4.1. The workflow, as well as the particular component, has numerous dependencies on libraries and other software, which is either very complex to install and procure or not open-source.

Within the activities of Task 3.1, the 3DAirQualityPrediction component is currently set up as a Singularity container and is operational on a virtual machine provided by PSNC. Native installation of the component on the two systems and therefore its benchmarking is currently in progress.

3.2.3 Social Networks Pilot

3.2.3.1 Current status of the workflow

The workflow of the social networks pilot is presented in Figure 3.8. Current effort is focused on the Validation component, which is responsible for computing the histogram of eigenvalues of the social network test graph that currently numbers 990,000 nodes. The component has extreme computational requirements and at the same time, occupies a very large amount of memory, thus requiring more than 100 nodes in order to execute. The code is written in C++, makes use of the PETSc and SLEPc libraries, and has two parts: one that performs an analysis step on the graph and one that performs factorization and computes the histogram. Currently, only the second part is parallelized using MPI and OpenMP.

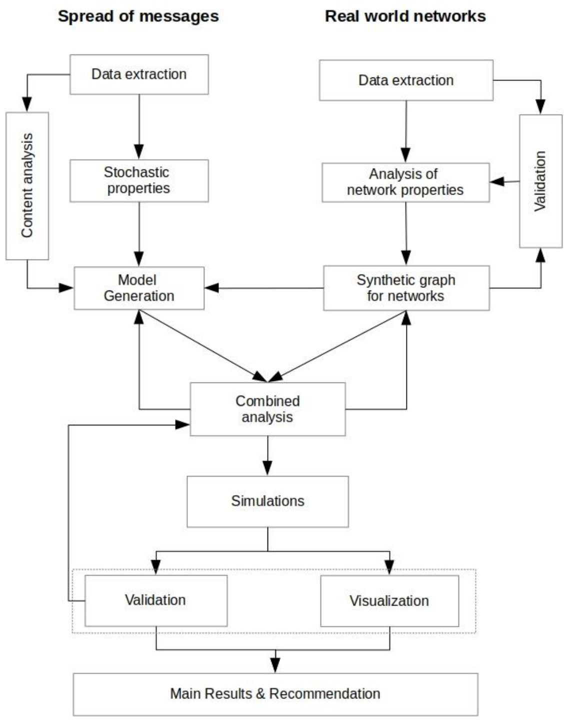


Figure 3.8: Social Networks Use Case - Workflow

3.2.3.2 Preliminary benchmarking findings

Preliminary runs of the Validation component on Hazelhen have focused on determining the right number of nodes and cores and an appropriate mix of MPI processes and OpenMP threads to achieve the seamless execution of the application for large graphs.

Figure 3.9 presents scalability results for a graph of 700,000 nodes on Hazelhen, for varying numbers of nodes, cores and mix of MPI processes/OpenMP threads. The component does not benefit from additional nodes and/or cores, as, by decreasing the number of nodes from 300 to 80, execution time for the factorization phase decreases. This reveals that the

Validation component is in fact communication-sensitive. However, using fewer nodes and cores, i.e. moving from 800 cores to 700 cores, can hurt the execution time; therefore, it is necessary to select the right configuration carefully.

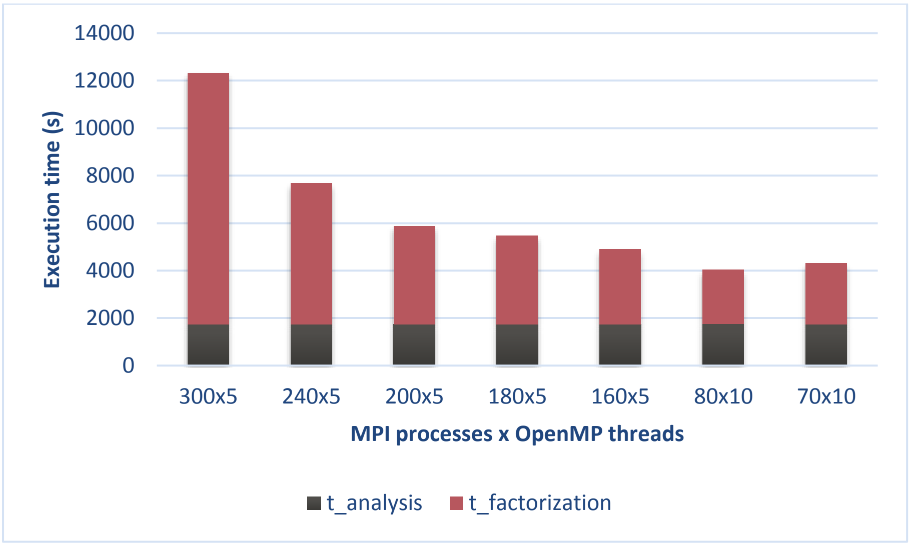


Figure 3.9: Scaling the Validation component on Hazelhen (HLRS) - Execution time for a graph with 700K nodes

Figure 3.10 presents the best achieved execution time on Hazelhen for various problem sizes, alongside with the core count in use for each problem size. The graph shows that, in order to be able to solve the problem of computing the histogram of the eigenvalues for large graphs, an additional 25-50 cores are required for every additional 50K nodes of the graph.

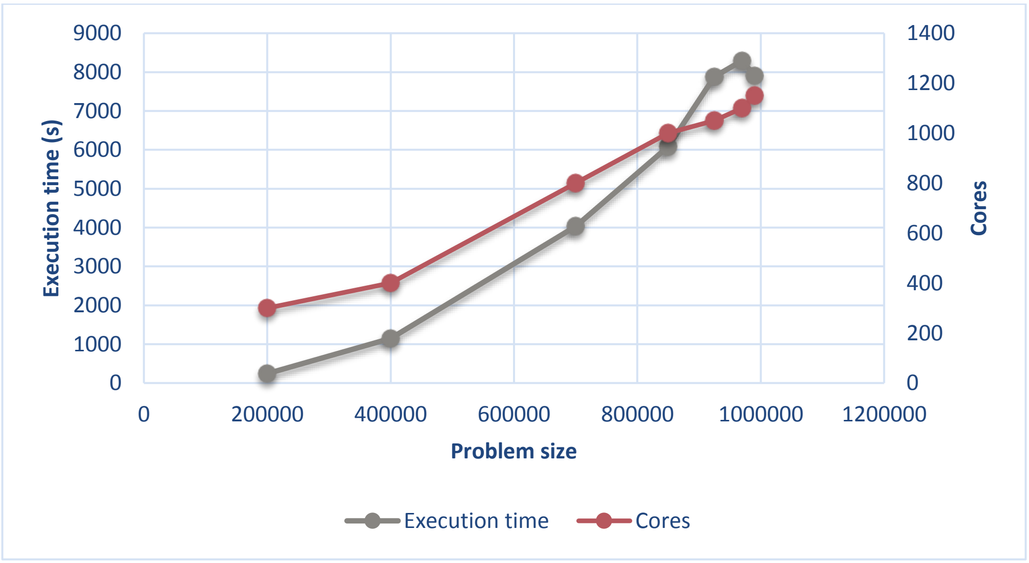


Figure 3.10: Scaling the problem size of the Validation component on Hazelhen (HLRS) - Execution time and required cores

In order to get a better idea of the scalability of the Validation component, we use a smaller input graph of 50,000 nodes, which allows the execution of the component on a single node.

Figure 3.11 shows the execution time on a single node of Hazelhen using various configurations of MPI processes and OpenMP threads. The factorization phase scales up to 24 cores for all possible configurations, achieving however, at most a speedup of 8 on 24 cores. What is noteworthy is that the usage of more OpenMP threads is more beneficial against the usage of more MPI processes within the node, i.e. configurations with a single process and multiple OpenMP threads achieve lower execution times compared to configurations with multiple processes on Hazelhen.

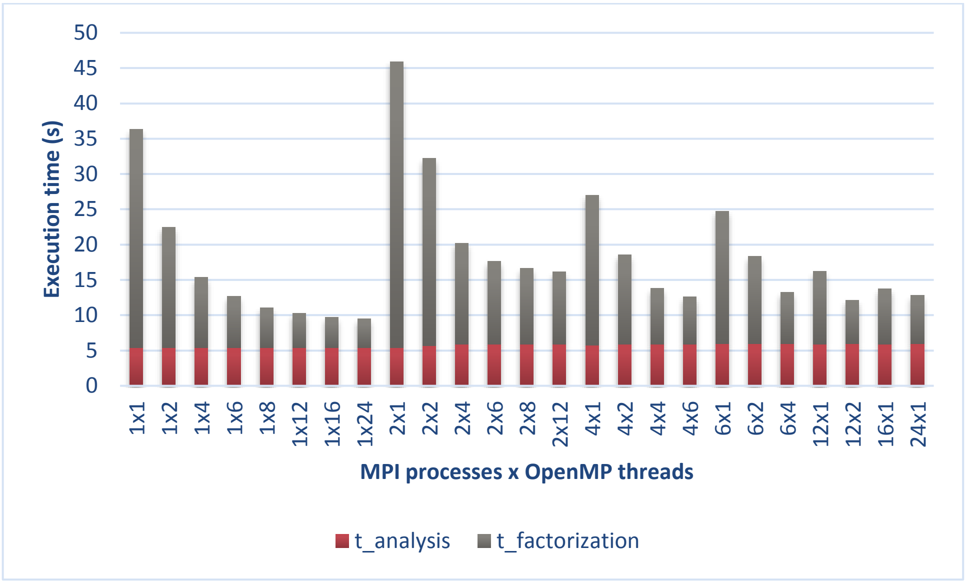


Figure 3.11: Scaling the Validation component on Hazelhen (HLRS) - Execution time for a graph with 50K nodes

Figure 3.12 presents the same experiment on a single node on Eagle. Unlike Hazelhen, the lowest execution times are achieved with more MPI processes, rather than more threads. Moreover, we can observe extreme peaks in execution time for specific configurations, which use 14 OpenMP threads (1x14, 2x14). The lowest execution time occurs when using the full node with 28 MPI processes, with a speedup of 6.5 over serial execution.

The scalability plots for the smaller graph does not offer sufficient insight as to what hinders higher speedups or why the two systems exhibit different scalability behaviour. Unfortunately, large scale runs on Eagle, using graphs of 400,000 nodes and 700,000 nodes, on similar configurations as on Hazelhen, with more than 30 nodes, all currently result in various errors that can be traced back to the PETSc library used by the module (segmentation faults, memory limits, InfiniBand timeouts). Further analysis of the module's requirements in memory and appropriate configuration and tuning of the PETSc library is necessary before we further benchmark and profile the application on Eagle.

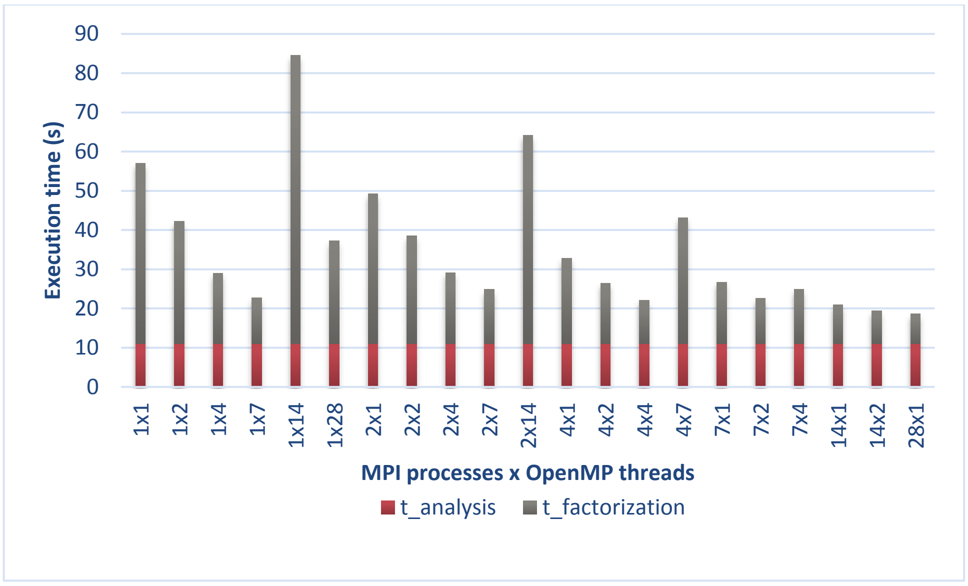


Figure 3.12: Scaling the Validation component on Eagle (PSNC) - Execution time for a graph with 50K nodes

4 Conclusions

4.1 General remarks & lessons learned

Deliverable D3.1 presents an initial definition of the HiDALGO benchmarking methodology. At this phase of the project, we have focused on the HiDALGO HPC infrastructure, and have targeted the compute-intensive parts of the HiDALGO pilots. We have based the HiDALGO benchmarking methodology on the existing HiDALGO infrastructure, a number of available tools, best practices drawn from our experience with HPC systems and applications, and the current status of the HiDALGO pilot applications. We have focused on defining a generic, systematic, reproducible, and interpretable methodology for collecting benchmarking information from the HiDALGO applications, and a systematic way of storing benchmarking results.

We have applied the basic steps of this methodology in full for the Migration pilot and have kick-started the benchmarking efforts for the Air Pollution and Social Networks pilots. This initial experimentation and benchmarking have helped us identify various major and minor issues in procuring and/or benchmarking the HiDALGO pilots and has significantly impacted the definition of the HiDALGO methodology.

4.2 Next steps

Next steps regarding benchmarking within HiDALGO include, but are not limited to, the following:

- Resolve pending issues with installation and procurement of all three HiDALGO pilots.
- Apply the HiDALGO benchmarking methodology to the HPC modules of all three HiDALGO pilots across HiDALGO infrastructure.
- Refine the HiDALGO benchmarking methodology to systematically collect more metrics of interest and gain further insight to the performance aspects of applications.
- Extend the HiDALGO benchmarking methodology to include HPDA infrastructure and HPDA modules of the HiDALGO pilots.
- Automate access to HiDALGO benchmarking results, exploiting the pre-defined repository structure.

References

[1] "CrayPAT". *Nersc.Gov*, 2019,http://www.nersc.gov/users/software/performance-and-debugging-tools/craypat/. Accessed on Feb 2019.

[2] "Extrae - HLRS Platforms". *Wickie.Hlrs.De*, 2019,https://wickie.hlrs.de/platforms/index.php/Extrae. Accessed Feb 2019.

[3] "Extrae". *Barcelona Supercomputing Center*, 2019,https://tools.bsc.es/extrae.Accessed on Feb 2019.

[4] "Home | Intel® Advisor". *Software.Intel.Com*, 2019,https://software.intel.com/en-us/advisor. Accessed on Feb 2019.

[5] "Intel® Inspector". Software.Intel.Com, 2019,https://software.intel.com/en-us/intel-inspector. Accessed on Feb 2019.

[6] "PAPI". *Icl.Cs.Utk.Edu*, 2019, http://icl.cs.utk.edu/papi/index.html. Accessed on Feb 2019.

[7] "Paraver: A Flexible Performance Analysis Tool". *Barcelona Supercomputing Center*, 2019,https://tools.bsc.es/paraver. Accessed on Feb 2019.

[8] "Vampir - HLRS Platforms". *Wickie.Hlrs.De*, 2019,https://wickie.hlrs.de/platforms/index.php/Vampir. Accessed on Feb 2019.

[9] Groen, D., 2018, June. Development of a multiscale simulation approach for forced migration. In *International Conference on Computational Science* (pp. 869-875). Springer, Cham.

[10] S. Andersson, "Using Perftools for threaded and hybrid codes", HLRS, Stuttgart, 2011.

[11]Saviankou, Pavel. "Cube 4.X Download". *Scalasca.Org*, 2019,http://scalasca.org/software/cube-4.x/download.html. Accessed on Feb 2019.

[12] W. Yang, "Using CrayPat", NERSC Oakland Scientific Facility, 2012.