TALP: A lightweight tool to unveil parallel efficiency of large-scale executions

Victor Lopez
victor.lopez@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Guillem Ramirez Miranda
guillem.ramirez.miranda@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Marta Garcia-Gasulla
marta.garcia@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

ABSTRACT

This paper presents the design, implementation, and application of TALP, a lightweight, portable, extensible, and scalable tool for online parallel performance measurement. The efficiency metrics reported by TALP allow HPC users to evaluate the parallel efficiency of their executions, both post-mortem and at runtime. The API that TALP provides allows the running application or resource managers to collect performance metrics at runtime. This enables the opportunity to adapt the execution based on the metrics collected dynamically. The set of metrics collected by TALP are well defined, independent of the tool, and consolidated. We extend the collection of metrics with two additional ones that can differentiate between the load imbalance originated from the intranode or internode imbalance. We evaluate the potential of TALP with three parallel applications that present various parallel issues and carefully analyze the overhead introduced to determine its limitations.

KEYWORDS

Performance and optimization, Performance Monitoring

ACM Reference Format:
Victor Lopez, Guillem Ramirez Miranda, and Marta Garcia-Gasulla. 2021. TALP: A lightweight tool to unveil parallel efficiency of large-scale executions. In *PERMAVOST '21.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The current HPC scenario involves growing systems targeting Exascale, a diversity of architectures, and applications from different scientific fields struggling to use the ever-increasing number of resources efficiently. But the efficient use of HPC resources is fragile. It depends on several factors: Hardware, system software, application code, and input set, to name the most relevant.

Achieving an acceptable efficiency is crucial when running large scale executions, as a decrease in the parallel efficiency can lead to losing large amounts of computational power. A simulation running on 10.000 cores with a parallel efficiency of 80% is not using 200 cores.

For users, the only way to ensure that their execution is running efficiently is through performance analysis. But, in general, they cannot afford to analyze every production run they need to do. We propose a tool with the following characteristics. First, it is easy to use; no previous HPC expertise is necessary to use it, neither to modify the application. Second, it is lightweight, with minimal added overhead; there is no other way to get users to use a tool in production simulations. A third aspect is that it reports well defined, relevant, and easy to understand metrics, the POP metrics [21]. Finally, it reports two additional metrics, providing information on the load imbalance source, whether it originates in the load between nodes or inside the computational nodes.

We do not aspire to substitute detailed performance analysis; on the contrary, we want to claim that it is indispensable and essential. We envision TALP as a probe that will raise the warning when further analysis is necessary.

TALP reports the collected metrics at the end of the execution for the user to decide if they are acceptable, she must change the configuration to run efficiently, or a more detailed performance analysis is necessary. Additionally, TALP offers this information at runtime through an API; it can be used by resource managers, system software, or applications implementing dynamic autotuning.

The main contributions of this paper are i) Specification, design, and implementation of TALP. ii) Offer a report of the POP metrics without obtaining a detailed trace. iii) Provide an API to consult this information at runtime. iv) The definition of two new metrics to distinguish intra and internode load imbalance.

2 RELATED WORK

There is a great variety of profiling and tracing tools for performance analysis; Score-P [15] is a project that tries to unify some of them. In general, these tools must choose between detail in the collected data and performance or overhead. Extrae [19], Scalasca [11], and Vampir [2], for example, are trace-based tools that allow the analyst to reconstruct in detail the program's execution. TAU[20] is a very flexible tool-set that can change from trace-based or profiling.

Some of the profiling tools that we can find are ompP [7] for OpenMP programs, mpiP for MPI or gprof [14]. Usually, these tools output a lot of information and measurements challenging to manage by a non-expert user.

With TALP, we aim at being a very light-weight tool. Our target is that it can be used in production runs or even part of the system software stack. Moreover, the metrics that TALP collects and summarizes are well-defined, well known, and able to capture parallel execution's fundamental characteristics.

PERMAVOST '21, June 25, 2021, Stockholm, Sweden
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

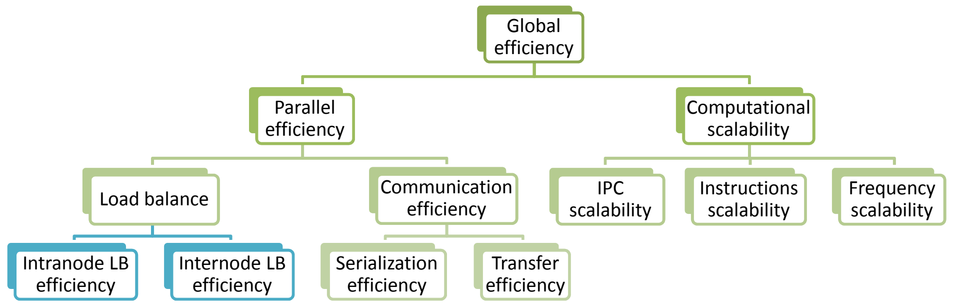


Figure 1: POP metrics

2.1 POP Efficiency metrics

The Center of Excellence (CoE) for Performance Optimization and Productivity (POP) [5] has designed a methodology for performance analysis and is promoting its utilization in the HPC community, more details on the methodology can be found in previous work [4, 21] and on their web page. Each metric they propose is designed to capture a fundamental characteristic of the execution, and is the starting point for a more detailed analysis.

For completeness, we describe here the current POP metrics to justify and put in context our proposal of extending these metrics with two new ones. One of the main characteristics of the POP metrics is that they are organized in a hierarchy. Each factor can be computed as the product of its child metrics, in Figure 1 we can see the hierarchy of efficiency metrics (in green).

The metrics named *scalability* are computed based on a baseline run to determine its relative scalability. As our work is based on runtime measurements, we do not have a baseline, therefore, we set aside the *Computation scalability* and its child metrics.

These efficiency metrics are based on the simplification of a process into two states: The state in which it is performing computation, is called *Useful* (blue) and the state in which it is not performing computation, e.g., communicating to other processes, is called *Not useful* (red).

We call $P = \{p_1, \ldots, p_n\}$ the set of MPI processes and n the number of MPI processes. For each MPI process p we define the set $U_p = \{u_1^p, u_2^p, \ldots, u_{|U|}^p\}$ of the time intervals where the application is performing useful computation (the set of the blue intervals). We define the sum of the durations of all useful time intervals in a process p as shown in Equation 2. Similarly we can define $\overline{U}_p$ and $D_{\overline{U}_p}$ for the red intervals. We also define the *elapsed time* E as shown in Equation 1.

$$E = max_{p=0}^{n} D_{U_p} + D_{\overline{U}_p} \tag{1}$$

$$D_{U_p} = \sum_{U_p} \blacksquare = \sum_{j=1}^{|U_p|} u_j^p \tag{2}$$

$$PE = \frac{\sum_{U_p} \blacksquare}{E * n} \tag{3}$$

$$LB = \frac{\sum_{i=1}^{n} D_{U_i}}{n * \max_{i=1}^{n} D_{U_i}} \tag{4}$$

$$CE = \frac{\max_{i=1}^{n} D_{U_i}}{E} \tag{5}$$

The Parallel efficiency indicates the amount of time that is being lost due to the parallelization of the code. Or, what is the same, the ratio between the time that is being used for useful computation and the total consumed CPU time. As we said the *Parallel efficiency PE* can be computed as the product of the *Load balance LB* and *Communication efficiency CE* and is defined as shown in Equation 3.

The *Load balance* measures the efficiency loss due to different loads (useful computation) for each process. And the *Communication efficiency* the time spent in communication that is not due to Load imbalance. Both metrics are defined in Equations 4 and 5

The *Communication Efficiency* is the product of the Transfer and the Serialization but as these two metrics cannot be computed online we will not enter into their details.

2.2 DLB library

TALP is integrated within the Dynamic Load Balancing (DLB) library. DLB is a framework that aims at improving the performance of parallel applications and is designed in a modular way. One of the main non-functional requirements of DLB is transparent and non-intrusive for the application and user. To achieve this, DLB is integrated with MPI, OpenMP, and OmpSs.

Additionally, it also offers a user-level API for fine-tuning or for advanced users. The current version of DLB include three modules:

- **LeWI** (Lend When Idle [8, 9]): Dynamically changes the number of threads used by the shared memory programming model to utilize the node's computing resources better.
- **DROM** (Dynamic Resource Ownership Management [6]): Allows a resource manager or the application to change the distribution of threads among processes to maximize the performance or the efficiency.
- **TALP** (Tracking Application Life Performance): Measures the parallel efficiency achieved post-mortem and at run time of a parallel execution and presented in this paper.

3 TALP

3.1 Implementation

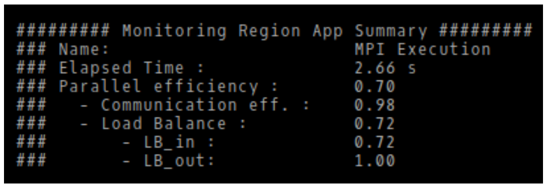
TALP is implemented within DLB as a dynamic library, this allows to use its functionalities by dynamically loading the library, and operating completely transparent to the user.

TALP is a portable, extensible, lightweight, and scalable tool for parallel performance measurement. TALP implements the well defined and established POP metrics and offers an API to consult them during the execution. The API can be used by the application or other resource managers or job schedulers.

The basis of the implementation intercepts the MPI calls and accounts for the time spent by each MPI process doing useful computation or communication. It also takes into account the number of threads that are being used in case the applications use a hybrid programming model.

3.2 Running an MPI application with TALP enabled

TALP is able to collect metrics of MPI applications at run time without requiring to modify or recompile the application as long as the system provides a mechanism to preload or override libraries when running an application. For instance, the dynamic linker `ld` in the Linux kernel allows preloading a list of objects with the `LD_PRELOAD` environment variable. By dynamically preloading the DLB library before the MPI symbols, the TALP module is able to intercept the MPI calls and gather all the needed metrics.



```
######### Monitoring Region App Summary #########
### Name:                           MPI Execution
### Elapsed Time :                  2.66 s
### Parallel efficiency :           0.70
###   - Communication eff. :        0.98
###   - Load Balance :              0.72
###       - LB_in :                 0.72
###       - LB_out:                 1.00
```

Figure 2: Summary of efficiencies

DLB should also be configured to initialize the TALP module with the option `--talp`, and the option `--talp-summary=app` to print an application summary at the end of the execution. This summary will include all the metrics explained in section 2.1, an example of this summary reported by TALP is shown in Figure 2.

Listing 1 shows an example of how to run any MPI application with TALP. Sometimes, the `mpirun` command can run some MPI functions and those must not be intercepted by DLB. For this reason, the `LD_PRELOAD` variable is better set only for the application and not for the `mpirun` command.

Listing 1: Running an application with TALP

```
export DLB_ARGS="--talp --talp-summary=app"
mpirun env LD_PRELOAD="$DLB_LIBS/libdlb_mpi.so" ./app
```

3.3 User API

In the previous section, it has been shown how TALP can obtain metrics of any MPI application for the entire execution. If application users decide to obtain TALP metrics for one or more delimited part of the application, they can do so by using the TALP user interface.

Users can define as many user-defined Monitoring Regions as they like, each region can be started and stopped as many times as necessary, and multiple regions can be nested in any form. The essential functions are the following and an example of utilization can be found in Listing 2:

- `dlb_monitor_t* DLB_MonitoringRegionRegister` Register a new monitoring region with a unique name and obtain a handler, or obtain a previously registered handler if a monitoring region exists with that name.
- `void DLB_MonitoringRegionStart` Start the monitoring region. Define the beginning of the delimited region.
- `void DLB_MonitoringRegionStop` Stop the monitoring region. Define the ending of the delimited region.

Listing 2: Monitoring Region usage

```
#include <dlb_talp.h>
...
// Register a new region or obtain an existing handler
dlb_monitor_t *monitor = DLB_MonitoringRegionRegister("Region name
    ");

// Start TALP monitoring region
DLB_MonitoringRegionStart(monitor);
...
// Stop TALP monitoring region
DLB_MonitoringRegionStop(monitor);
```

For each user-defined Monitoring Region, TALP will print a summary at the end of the execution as long as the option is provided. This way, the user will have the metrics for the entire execution as well as the metrics for each region as they were isolated.

Some metrics can also be obtained from within the application before the end of the execution. The monitor handle is defined as a struct in the `dlb_talp.h` header file and some of the data contained can be publicly accessed. Finally, if only a report is required, TALP provides another function to print a report of that specific monitor. Both ways can be seen in Listing 3.

Listing 3: Obtaining the data from the Monitoring Region

```
// Print a report by standard output
DLB_MonitoringRegionReport(monitor);

// Manually obtain some metrics from the monitor
int64_t elapsed = monitor->elapsed_time;
int64_t elapsed_useful = monitor->elapsed_computation_time;
float comm_eff = (float)elapsed_useful / elapsed;
```

3.4 Efficiencies

Additionally to the metrics explained in Section 2.1 we propose two new metrics that have been implemented in TALP to demonstrate its potential. The proposed metrics are child metrics of the Load balance, as can be seen in Figure 1. They consist in separating the Load balance that is achieved among the different nodes: *LB internode* LB_α, and the one that is achieved inside the nodes: *LB intranode* LB_β. We define N as the number of nodes used and k as the number of processes per node ($k = n/N$). We also need to extend the previous definition of each process as $p_{j,i}$ where j is the node where the process is running and i the rank of this process within the node, and accordingly the definition of $D_{U_{j,i}}$.

$$LB_\alpha = \frac{\sum_{j=1}^{N} \sum_{i=1}^{k} D_{U_{j,i}}}{\max_{j=1}^{N} \sum_{i=1}^{k} D_{U_{j,i}} * N} \quad (6)$$

$$LB_\beta = \frac{\max_{j=1}^{N} \sum_{i=1}^{k} D_{U_{j,i}}}{max_{i=1}^{n} D_{U_i} * k} \quad (7)$$

The LB internode LB_α represents the load balance between the different nodes. The load of a node is the sum of the useful computation of all the processes on the node. Thus, the LB among nodes is computed as the avg/max analogous to the load balance among processes.

The LB intranode LB_β represents the load balance of the most loaded node, once we balance the load inside the nodes the limiting node will be the most loaded one. For this, we compute LB_β as the ratio between the best situation and the current one, as this is the efficiency that we are losing. The best situation would be to have the most loaded node perfectly balanced, that is the load of the most loaded node divided by the number of processes per node, and the current situation is the most loaded process from all of them.

4 EVALUATION

In this section, we present the results from evaluating the performance and validation of TALP.

4.1 Environment and methodology

All the experiments have been performed in Marenostrum4. Marenostrum4 is a tier1 platform based on Intel Xeon Platinum with 3456 nodes, and each node has two sockets with 24 cores each and 96GB of main memory. Its nodes are connected through a 100 Gbit/s Intel Omni-Path network.

We use three different use cases for the evaluation:

PILS is a microbenchmark developed to represent applications with different patterns of load imbalance. It is implemented in C and parallelized using MPI, OpenMP, and OmpSs. We use PILS to do a detailed analysis of the granularity that TALP can support without adding relevant overhead. Also, to validate its accuracy when measuring load imbalance.

Alya is a simulation code for high performance computational mechanics. It can solve different physics and is widely used for production simulations. Alya is developed using Fortran and parallelized with MPI, OpenMP and, partially ported to GPUs. We use a state-of-the-art combustion use case consisting of 100000 elements.

CP2K is an open source molecular dynamics software package to perform atomistic simulations of solid-state, liquid, molecular, and biological systems. It is aimed at massively parallel systems and state-of-the-art molecular dynamics simulations. It is used by many research groups for their simulations, it is written in Fortran and parallelized with MPI [13, 16]. We use the H2O input set, a benchmarking and flexible input that allows modifying the computational load by adjusting the number of molecules.

The selection of the use cases is based on the following; on the one hand, PILS offers flexibility to evaluate in detail the limitations of TALP and validate it. On the other hand, Alya and CP2K are HPC production codes, and both belong to the UEABS [18] (Unified European Applications Benchmark Suite). From the description of the UEABS, the benchmark suite codes are *"[...] scalable, currently relevant and publicly available[...], of a size which can realistically run on large systems and maintained into the future"*. Moreover, from previous works, we know that, in particular input sets, Alya can present a severe load imbalance problem [10] while CP2K shows a communication issue that can limit its scalability [1]. With this selection, we covered the two main metrics we want to demonstrate with TALP: Load balance and Communication efficiency.

All the codes have been compiled using the compiler suite Intel 2017.4 and executed with Intel MPI 2017.4.

To validate our approach with Alya and CP2K, we compare the efficiency metrics collected by TALP with the ones obtained through the use of the performance toolbox from BSC [3]. This workflow includes the following steps: First, generate the traces with Extrae [19], then select the Focus of Analysis and cut it using Paraver [17]. Finally, compute the efficiency metrics with the *Basic Analysis* tool and the Dimemas [12] simulator. This process is explained in detail by Wagner et al. [21].

4.2 PILS

We divide the evaluation of TALP using PILS into two parts. The first one is to determine the limitations of TALP in terms of the granularity it can handle without introducing a relevant overhead. The second one is to validate the Load Balance metric by generating a given load balance with PILS.

4.2.1 Overhead Study. To evaluate the overhead introduced by TALP, we use PILS with the code structure shown in Listing 4, the loads array contain a value for each MPI process allowing to set a determined load imbalance, the duration parameter is the duration in microseconds of the computational task performed.

Listing 4: PILS code structure

```
int me;
MPI_Comm_rank(MPI_COMM_WORLD, &me)
for (i=0; i++; i<loops){
    for (j=0; j++; j<loads[me]){
//Start TALP monitoring region
        compute_task(duration);
        MPI_Allreduce(...)
        MPI_Barrier(MPI_COMM_WORLD)
//Stop TALP monitoring region
    }
}
```

For this experiment, the PILS variables are set to the following values shown in Listing 5. In order to depict the granularity in a meaningful metric we computed the MPI calls per millisecond based on the duration of the computation between MPI calls as: $MPI\ calls\ per\ ms = \frac{1}{duration} * 2$. We perform each experiment 20 times.

Listing 5: PILS variables for overhead study

```
loops=10000
loads=1
duration={1,5,10,20,40,50,100,200,300,400,500,750,1000,1500,2000}
```

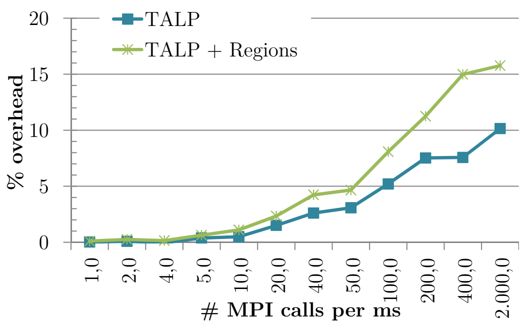


Figure 3: Overhead study with PILS

In Figure 3 we can see the overhead introduced by TALP. The overhead is computed as $\%overhead = ((\frac{t\_talp}{t\_orig}) - 1) * 100$; where t_talp is the elapsed time of the whole execution when using TALP and t_orig the elapsed time of PILS with the same settings without TALP. In the x axis we can see the MPI calls per millisecond performed, and the two series correspond to the use of TALP to measure the whole run, while the `TALP + Regions` series corresponds to adding a TALP region to measure that starts at line 5 and stops at line 8. This means that it has the same frequency as the MPI calls.

We can observe that the overhead of TALP is below 5% for MPI call frequencies below 50 MPI calls per millisecond, and below 10% when doing 1.000 MPI calls per ms (one MPI call per us). When using monitoring regions and starting-stopping them at the same frequency as MPI calls the overhead is higher, but still below 5% when doing 25 MPI calls per millisecond or less.

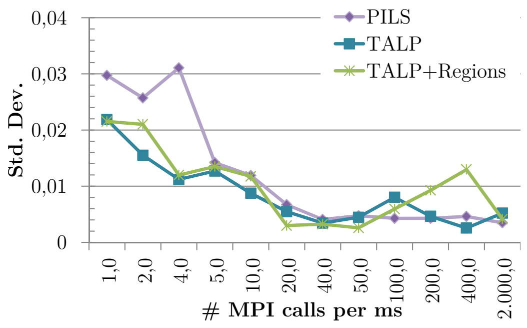


Figure 4: St. Dev. overhead study with PILS

In Figure 4 we show the standard deviation between the 20 samples of the same run, to understand if noise or variability is added with the use of TALP. In the x axis is represented the frequency of MPI calls and the three series correspond to the execution without TALP (labeled *PILS*), execution of PILS with TALP using `LD_PRELOAD` (labeled *TALP*) and the execution of PILS with TALP enabled and a monitoring region that start-stops every iteration (labeled *TALP + Regions*).

This experiment shows that there is no variability added by TALP nor the monitoring regions, the different versions executed present a standard deviation of the same order of magnitude.

4.2.2 Validation Study. For the validation study we use the same PILS code structure but setting the variables to the values shown in Listing 6. We run the experiments with 2 MPI processes and each process with a different load set by the variable `load`. Based in these loads we compute the theoretical load balance of the execution. We compute the error of the load balance measured by TALP as $\%\ error = |theoretical\_LB - measured\_LB| * 100$. As Load balance takes a value between 0 and 1.

Listing 6: PILS variables for LB validation study

```
loops=10000
loads={<1,99>, <10,99>, <25,75>, <40,60>, <55,45>, <50,60>}
duration={1,10,100,1000}
```

In Figure 5 we see the % error of the load balance measured by TALP for different granularities. We can see that for a frequency of MPI calls below 100 MPI calls per millisecond the error is below the 1% in all the cases. If there are 1000 MPI calls per millisecond the error is below 3% for all the loads. The trend is that the error is slightly higher for well balanced executions, this makes sense as for well balanced runs a small variation can change the load balance. From this experiment, we can conclude that TALP offers a precise measurement of the Load Balance even in extreme cases where the overhead is almost 10%.

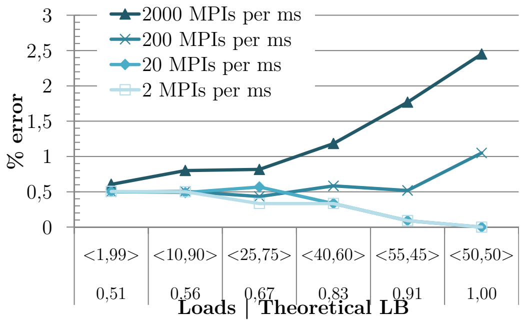


Figure 5: % error measuring LB with PILS

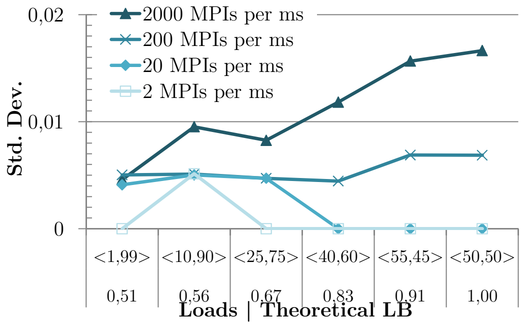


Figure 6: St. Dev. of the LB measured with PILS

On Figure 6 we can see the standard deviation computed for the different runs of the same kind. We can observe that the executions with a high frequency of MPI calls (ie. 1000 MPI calls per millisecond) there is more variation than for runs with a lower frequency. Analyzing the data collected in detail we observe that this variation in the measurement is because some of the executions obtained a perfect measured Load Balance, while other showed signs of being affected by noise, this means that the error of TALP does not come from overhead or error in measure but from noise in the execution.

We have demonstrated that even for extreme situations with a high frequency of MPI calls the Load balance measured is within a 3% of error.

4.3 CP2K

4.3.1 Time. The first study of TALP based on CP2K consists in measuring the difference in the execution time between using TALP or not, because our primary goal is to provide a lightweight tool that can be used in production simulations. The elapsed time is computed as the average of 5 different executions solving 10 timesteps.

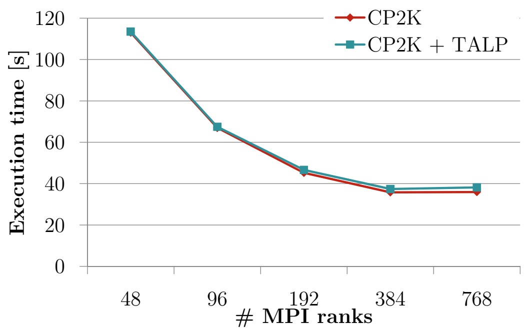


Figure 7: Execution time of CP2K up to 16 nodes

In Figure 7, we can see the execution time of a CP2K simulation with and without TALP when scaling up to 16 nodes of Marenostrum4 (768 MPI ranks). We can observe that there is not a relevant difference in the execution time.

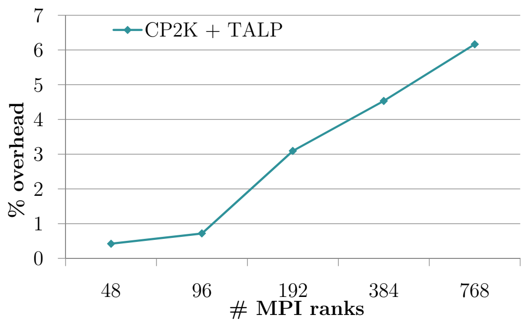


Figure 8: % overhead of TALP running CP2K up to 16 nodes

On Figure 8 we find the % overhead computed as before, we can see that it is below a 7% in all the cases and it increases slowly with the number of MPI processes.

4.3.2 Efficiencies. To evaluate the efficiencies measured with TALP we compare the measures obtained by TALP with the efficiencies obtained by getting a trace, cutting it and passing the cut through the Basic analysis tool.

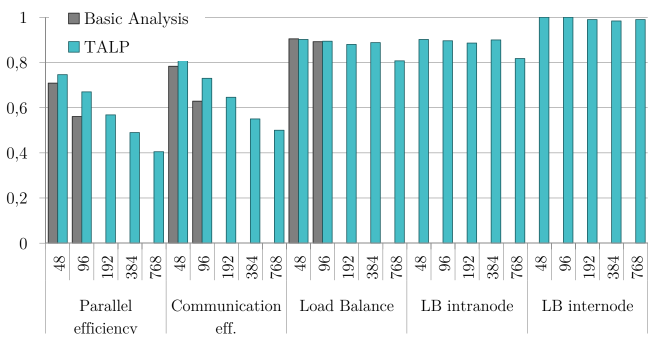


Figure 9: CP2K efficiencies with TALP

In Figure 9 we can see the efficiencies obtained by the execution of CP2K up to 16 nodes. In the x axis are shown the different efficiency metrics and the number of MPI ranks used to run the simulation. For the Basic analysis method, we only obtained traces for 48 and 96 MPI ranks because the traces when using more MPI ranks were too big to manage and process with the given tools (90GB for 192 MPI ranks, 253GB for 384 MPI ranks, and 612GB for 768 MPI ranks).

We can see that for the Load balance metric TALP obtains the same value as the Basic analysis tool. For the Communication and Parallel efficiency, the values obtained by TALP are slightly higher than the ones measured using Extrae + Basic analysis. This is due to the high amount of communication of this application, and the fact that the overhead of these tools is accounted as MPI time. The use of TALP in this case allows us to see that CP2K shows a good Load balance up to 768 MPI ranks. The Load balance is a bit worse intranode than the internode, this means that the load is distributed equally among the different nodes.

We also observe that the main factor that is limiting the scalability is the Communication efficiency, with 768 MPI ranks it achieves a value of 0.5 meaning that half of the computational resources used by the simulation are not being used to do useful computation.

4.4 Alya

With Alya we solve a combustion problem, we have two different input sets with the following characteristics:

Simple: Solves a simple chemical reaction, the number of equations solved is low, the computational load is low, the load imbalance is very high and there is an important intranode load imbalance.

Complex: Solves a complex chemical reaction, the computational load is high, the load imbalance is high.

In the case of Alya we add a monitoring region around the code computing the chemical reaction, as it is where the load imbalance appears.

4.4.1 Time. Analogous to the test with CP2K, with Alya we start measuring the overhead introduced in the execution. The elapsed time in Alya is computed per timestep, we run the simulation for 10 timesteps and average their duration, then we launch 5 independent runs and also average their average timestep duration.

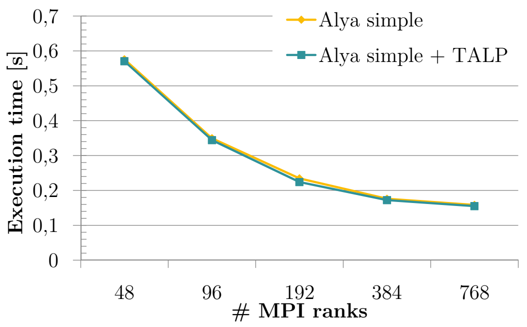


Figure 10: Execution time of Alya simple up to 16 nodes

We can find in Figure 10, the execution time when scaling the simple input set up to 768 MPI ranks, with and without TALP. We observe that there is no overhead introduced by TALP when scaling up to 16 nodes.

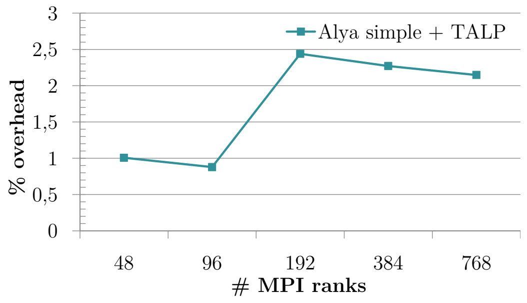


Figure 11: % overhead with TALP running Alya simple

On Figure 10 we can see the overhead measured as percentage and we see that it is always below 3%.

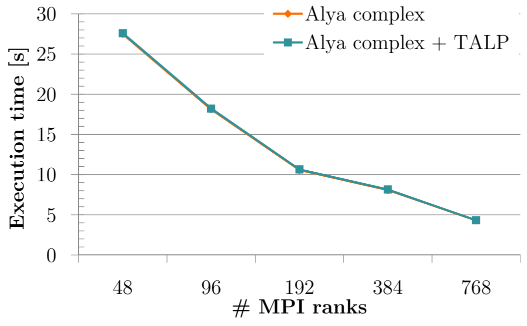


Figure 12: Execution time of Alya complex up to 16 nodes

In Figure 12 we depict the execution time of the simulation of the complex chemical reaction with Alya. We can observe that there is no relevant difference between the execution with and without TALP.

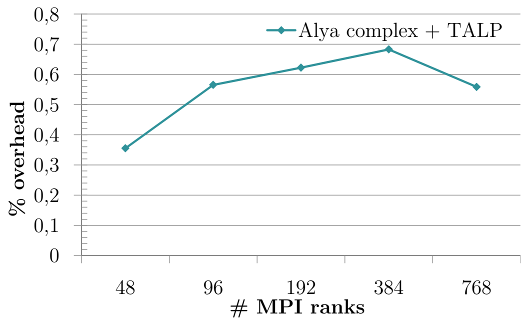


Figure 13: % overhead with TALP running Alya complex

On Figure 12 we can see the overhead for the complex simulation, and it shows that the overhead is always below 1%.

4.4.2 Efficiencies. To compare the efficiencies obtained by TALP with the Basic analysis tool, we follow the same method as previously, obtaining the trace, cutting it, and passing the cut through the Basic analysis tool. In this case, we have been able to obtain the traces for all the executions and the data obtained is shown in Figure 14. In the x axis we see the efficiency metrics obtained with TALP and the Basic analysis tool and the different MPI ranks used to run.

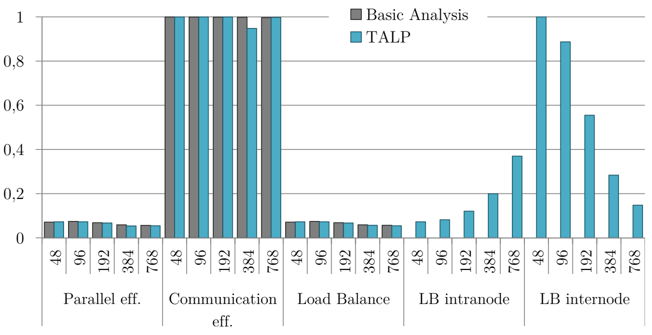


Figure 14: Alya simple efficiencies with TALP and Basic analysis

First we observe that for all the metrics the values obtained by TALP are very similar to the ones computed through the traces. We can conclude that the metrics collected by TALP are correct.

But we also observe that the parallel efficiency of this execution is extremely low, and the main issue limiting the scalability is the Load balance. The Communication efficiency is almost perfect. Looking in detail at the Load Balance metrics we can see that the intranode Load balance is very low when running with few MPI processes and increases as we increase the number of MPI ranks, this probably means that there are a few MPI processes very loaded due to the partition of the domain, and when the problem is divided in more MPI ranks this load is concentrated in one or few nodes, making the internode Load balance to decrease as we add more nodes to the execution. The effect of the intranode imbalance going up is due to the fact that the most loaded processes are in the same node. From the data collected with TALP we can conclude that this simulation is a good candidate to be executed with LeWI to load balance within a node, and that probably using a round robin distribution of MPI ranks would be beneficial for the execution also, because this would distribute the load among the different nodes.

In Figure 15 we can see the efficiencies obtained by the complex simulation of Alya when running up to 768 MPI ranks. We can see that the efficiencies computed by TALP are an exact match for those computed with the basic analysis. Looking into more detail in the efficiency metrics of this simulation we can say that it has a bad parallel efficiency and that the communication efficiency is almost perfect. The main issue limiting the scalability of this simulation is the load balance and in particular the intranode load balance. Although the internode load balance is not good, this simulation would not benefit so much from running with a round robin distribution of MPI processes among nodes as the main problem is within the most loaded node.

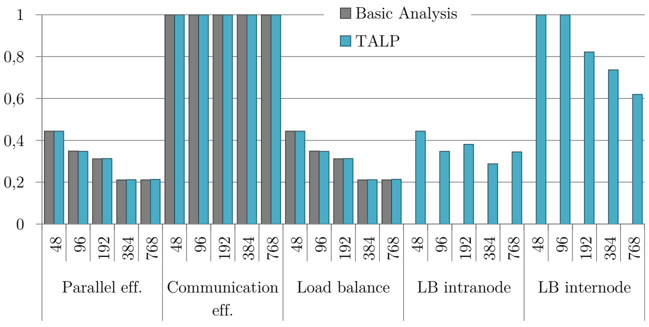


Figure 15: Alya complex efficiencies with TALP and Basic analysis

5 CONCLUSIONS AND FUTURE WORK

In this paper we present TALP, a transparent to the application, lightweight, and scalable tool to measure the parallel efficiency of MPI applications. It can be utilized by inexperienced users just by adding a couple of lines to the submission script and it will report the POP metrics about parallel efficiency. These metrics capture fundamental behaviour of the execution, indicating if the run has an acceptable parallel performance, or if not which are the main factors limiting it.

Additionally it offers an API for advanced users, that allow measuring specifics part of the code in case the user is interested in having a more detailed report. The API can also be used by resource managers or auto-tuning applications that want to adapt the execution dynamically at runtime. The API gives information that can be used to decide on requesting more resources or on the contrary releasing them to achieve a target parallel efficiency.

We show that the overhead added is not relevant while the frequency of MPI calls is below 50 MPI calls per millisecond, this corresponds to bursts of useful computation of 20 μseconds. We demonstrate the use of TALP with and without the API with two widely used HPC scientific applications.

Finally, we introduce two new metrics that can give relevant information on the Load balance efficiency and how to address it. The two new metrics differentiate between the load balance inside the nodes and across nodes, telling us if it is worth to use the DLB library to load balance, and to launch he application with a round robin distribution of processes across nodes.

As future work we plan to add the option of collecting hardware counters during the execution, and also to measure the frequency of MPI calls done by the application, in case that it is above the threshold of acceptable overhead, we can either deactivate TALP or emit a warning to the users advising them to take the overhead into account.

REFERENCES

[1] Fabio Banchelli, Kilian Peiro, Andrea Querol, Guillem Ramirez-Gargallo, Guillem Ramirez-Miranda, Joan Vinyals, Pablo Vizcaino, Marta Garcia-Gasulla, and Filippo Mantovani. 2020. Performance study of HPC applications on an Arm-based cluster using a generic efficiency model. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 167–174.

[2] Holger Brunst, Daniel Hackenberg, Guido Juckeland, and Heide Rohling. 2010. Comprehensive performance tracking with vampir 7. In *Tools for High Performance Computing 2009*. Springer, 17–29.

[3] BSC tools for performance analysis. [n.d.]. https://tools.bsc.es.

[4] Marc Casas, Rosa Badia, and Jesús Labarta. 2008. Automatic analysis of speedup of MPI applications. In *Proceedings of the 22nd annual international conference on Supercomputing*. 349–358.

[5] CoE Performance Optimization and Productivity (POP). [n.d.]. https://pop-coe.eu/.

[6] Marco D'Amico, Marta Garcia-Gasulla, Víctor López, Ana Jokanovic, Raül Sirvent, and Julita Corbalan. 2018. DROM: Enabling Efficient and Effortless Malleability for Resource Managers. In *Proceedings of the 47th International Conference on Parallel Processing Companion*. ACM, 41.

[7] Karl Fürlinger and Michael Gerndt. 2005. ompP: A profiling tool for OpenMP. In *International Workshop on OpenMP*. Springer, 15–23.

[8] Marta Garcia, Jesus Labarta, and Julita Corbalan. 2014. Hints to improve automatic load balancing with LeWI for hybrid applications. *J. Parallel and Distrib. Comput.* 74, 9 (2014), 2781–2794.

[9] Marta Garcia-Gasulla, Guillaume Houzeaux, Roger Ferrer, Antoni Artigues, Victor López, Jesús Labarta, and Mariano Vázquez. 2019. MPI+ X: task-based parallelisation and dynamic load balance of finite element assembly. *International Journal of Computational Fluid Dynamics* 33, 3 (2019), 115–136.

[10] Marta Garcia-Gasulla, Filippo Mantovani, Marc Josep-Fabrego, Beatriz Eguzkitza, and Guillaume Houzeaux. 2020. Runtime mechanisms to survive new HPC architectures: a use case in human respiratory simulations. *The International Journal of High Performance Computing Applications* 34, 1 (2020), 42–56.

[11] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. 2010. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 702–719.

[12] Sergi Girona, Jesús Labarta, and Rosa M Badia. 2000. Validation of Dimemas communication model for MPI collective operations. In *European Parallel Virtual Machine/Message Passing Interface UsersâĂŹ Group Meeting*. Springer, 39–46.

[13] Jürg Hutter, Marcella Iannuzzi, Florian Schiffmann, and Joost VandeVondele. 2014. cp2k: atomistic simulations of condensed matter systems. *Wiley Interdisciplinary Reviews: Computational Molecular Science* 4, 1 (2014), 15–25.

[14] Susan L Graham Peter B Kessler and Marshall K McKusick. 1982. gprof: a Call Graph Execution Profiler1. In *Proceedings of the Symposium on Compiler Construction, pp.–. Press, New York,,. Cited on p.. Greco, Gianluigi*. Citeseer.

[15] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. 2012. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*. Springer, 79–91.

[16] Thomas D Kühne, Marcella Iannuzzi, Mauro Del Ben, Vladimir V Rybkin, Patrick Seewald, Frederick Stein, Teodoro Laino, Rustam Z Khaliullin, Ole Schütt, Florian Schiffmann, et al. 2020. CP2K: An electronic structure and molecular dynamics software package-Quickstep: Efficient and accurate electronic structure calculations. *The Journal of Chemical Physics* 152, 19 (2020), 194103.

[17] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. 1995. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: transputer and occam developments*, Vol. 44. Citeseer, 17–31.

[18] Prace, UEABS. [n.d.]. https://repository.prace-ri.eu/git/UEABS/ueabs/.

[19] Harald Servat, Germán Llort, Kevin Huck, Judit Giménez, and Jesús Labarta. 2013. Framework for a productive performance optimization. *Parallel Comput.* 39, 8 (2013), 336–353.

[20] Sameer S Shende and Allen D Malony. 2006. The TAU parallel performance system. *The International Journal of High Performance Computing Applications* 20, 2 (2006), 287–311.

[21] Michael Wagner, Stephan Mohr, Judit Giménez, and Jesús Labarta. 2017. A Structured Approach to Performance Analysis. In *International Workshop on Parallel Tools for High Performance Computing*. Springer, 1–15.