Analyzing the Efficiency of Hybrid Codes

1st Judit Giménez
Technical University of Catalonia
Barcelona Supercomputing Center
Barcelona, Spain
0000-0002-2501-2791

2nd Estanislao Mercadal
dept. of Computer Science
Barcelona Supercomputing Center
Barcelona, Spain
0000-0002-1835-8671

3rd Germán Llort
dept. of Computer Science
Barcelona Supercomputing Center
Barcelona, Spain
0000-0002-7345-0841

4th Sandra Mendez
dept. of Computer Science
Barcelona Supercomputing Center
Barcelona, Spain
0000-0002-5793-1928

***Abstract*—Hybrid parallelization may be the only path for most codes to use HPC systems on a very large scale. Even within a small scale, with an increasing number of cores per node, combining MPI with some shared memory thread-based library allows to reduce the application network requirements. Despite the benefits of a hybrid approach, it is not easy to achieve an efficient hybrid execution. This is not only because of the added complexity of combining two different programming models, but also because in many cases the code was initially designed with just one level of parallelization and later extended to a hybrid mode. This paper presents our model to diagnose the efficiency of hybrid applications, distinguishing the contribution of each parallel programming paradigm. The flexibility of the proposed methodology allows us to use it for different paradigms and scenarios, like comparing the MPI+OpenMP and MPI+CUDA versions of the same code.**

***Index Terms*—Efficiency model, Hybrid parallelization, Scalability efficiency, Performance analysis**

I. Introduction

A large number of HPC codes are only based in MPI, but as the number of cores per node increases on the new architectures, extending the applications to a hybrid execution with MPI+*X* allows to reduce their network requirements compared with a pure MPI run. In the case of an architecture with accelerators, a hybrid approach is the only way to use all the available computing resources.

Even more important than using all the available resources is to make an efficient use of these resources. The BSC efficiency model [1] allows to diagnose how efficiently a parallel code is running with respect to some basic fundamental factors like load balance or data transfer, as well as pointing out which factor(s) may limit the application scalability. The efficiency model characterizes both the application and the platform (hardware and software stack) based on the computing regions and the time spent in the parallel runtime. While the efficiencies on the computations require using one of the executions as a base run (typically the smaller core-count run) and are computed as a scaling efficiency, the parallel efficiency allows to diagnose an isolated run based on the time spent in the parallel runtime, considered as the overhead paid to run in parallel. The original BSC model is described in detail in the Background section.

This model has been demonstrated to be very useful for MPI applications and in general for parallel applications that only exploit one level of parallelism. In the case of hybrid codes the current approach provides the analysis only at the hybrid level. But it is important to determine which of the components and which factors for these components are causing a higher loss of efficiency. This paper presents a solution to separate the contribution of each programming model when analyzing the efficiency of a hybrid parallel code.

The methodology we propose in this paper follows the philosophy of the BSC performance tools [3] targeting flexibility and simplicity. We maintain the small set of key factors of our initial model, characterizing causes of inefficiency that are common to all parallel paradigms instead of targeting specific inefficiencies of a given programming model. The proposed model enables to use the same methodology and metrics for different scenarios like MPI+OpenMP and MPI+CUDA; as well as for hybrid codes with a hierarchical approach, i.e., threads are blocked or idle during the MPI calls; and more dynamic codes with a task-based approach at the thread level, where the communications may be scheduled on any of the threads while the rest of the threads are computing.

The model has been sucessfully validated with controlled test cases where we can easily forecast what we want the model to report. We have also used the model to compare the executions of a well-known benchmark running with MPI+OpenMP and MPI+CUDA. Finally we describe the insight provided when assesing the performance of a real application that uses MPI for a distributed execution and accelerators to speed the computations.

The rest of this paper is organized as follows: Section 2 presents as background the BSC efficiency model, Section 3 describes the methodology and proposed model, Section 4 details the validation and experimentation, Section 5 presents related work, and finally Section 6 covers conclusions and future work.

DOI 10.1109/ISPDC51135.2020.00014

II. Background

The BSC efficiency model is based on a multiplicative speedup model [4]. The global efficiency is defined by two performance factors expressed in (1).

$$Global\_Eff = Par\_Eff \cdot Comp\_Eff \quad (1)$$

Where parallel efficiency (Par_Eff) exposes the inefficiency caused by the time spent in the parallel runtime and computation scalability efficiency ($Comp\_Eff$) characterizes inefficiencies scaling the computing regions.

To formulate the efficiencies, we denote T as the execution time, P as the number of MPI processes of a parallel application, and $Useful_i$ as the computation time outside the parallel runtime for the i process with $i \in [1, P]$.

Par_Eff defined in (2) is the percentage of time in $Useful$ computation, and it can also be expressed as the product of two efficiencies: load balance efficiency (LB_Eff) evaluates the distribution of computations across processes, and communication efficiency ($Comm\_Eff$) characterizes the communication time not caused by global unbalance between processes ((3) and (4)).

$$Par\_Eff = \frac{\sum_{(i=1)}^{P} Useful_i}{P \cdot T} = LB\_Eff \cdot Comm\_Eff \quad (2)$$

$$LB\_Eff = \frac{\frac{\sum_{(i=1)}^{P} Useful_i}{P}}{max(\{Useful_1, .., Useful_P\})} \quad (3)$$

$$Comm\_Eff = \frac{max(\{Useful_1, .., Useful_P\})}{T} \quad (4)$$

$Comm\_Eff$ captures two causes of communication time: 1) temporal unbalance or serializations that are compensated along time and 2) real waiting time due to data transfer. These can be measured with two metrics: serialisation efficiency (Ser_Eff) and transfer efficiency ($Transfer\_Eff$). To obtain these two sub-metrics, we use the Dimemas [5] simulator with an ideal network (zero latency and infinite bandwidth) to isolate the serialization efficiency from the transfer efficiency represented in (5) and (6) respectively.

$$Ser\_Eff = \frac{max(\{Useful_1, .., Useful_P\})}{T_{ideal}} \quad (5)$$

$$Transfer\_Eff = \frac{T_{ideal}}{T} = \frac{Comm\_Eff}{Serial\_Eff} \quad (6)$$

Using a similar approach, the computation scalability efficiency ($Comp\_Eff$) that characterizes inefficiencies scaling the computations can be decomposed in three factors: instructions scaling, IPC (instructions per cycle) scaling and clock frequency scaling allowing to identify the main source(s) of scaling problems [6].

This efficiency model has been adopted by the POP Center of Excellence [2] were it has been applied to analyse more than two hundred parallel codes. The Performance Optimisation and Productivity Centre of Excellence in HPC promotes best practices in performance analysis and parallel programming providing performance assesment and optimisation services for academic and industrial code(s) in all domains.

III. Hybrid model decomposition

As we have seen, the global efficiency of an application is characterized by the efficiency of the parallelization (parallel efficiency) and the efficiency scaling the computations (computation scalability). There is no need to decompose the computation scalability metric and it can be measured as in the original model. Our goal is to split the parallel efficiency and its first level components (load balance and communication efficiencies) between the two programming models. For clarity, the rest of this section describes the methodology using MPI and OpenMP as the two programming models of the hybrid code, despite the same efficiency model can be used with CUDA, OpenCL, POSIX Threads, etc.

The metrics at the hybrid level are computed like in the previous model, using as input a per-thread profile of the total time inside and outside the parallel run-times. To formulate the efficiencies we denote T as the execution time, P as the number of MPI processes, t as the number of threads per MPI rank , and $Useful_{i,j}$ as the computation time outside the parallel run-times for the thread j of process i with $i \in [1, P]$ and $j \in [1, t]$. In the hybrid scenario, the parallel efficiency is defined by (7) and similarly we can define the hybrid load balance and communication efficiencies.

$$Hybrid\_Par\_Eff = \frac{\sum_{(i=1,j=1)}^{P,t} Useful_{i,j}}{P * t \cdot T} \quad (7)$$

Figure 1 plots the hierarchy of metrics for an MPI + OpenMP hybrid code. The blue boxes correspond to the original model and the green boxes to the extended model proposed in this paper. The light blue lines that connect blue boxes with the green boxes identify new decompositions of previous factors. The extended model allows to compute each of the 3 efficiencies (parallel efficiency, load balance and communication efficiency) for the three programming paradigms (hybrid, MPI and OpenMP).

The 3-level hierarchy can be traversed through two orthogonal paths, classifying first by model factor (hybrid load balance vs. hybrid communication efficiency) or distinguishing first the impact of each programming model component (MPI vs. OpenMP). Despite these partial views are possible, our recommendation is to look at the nine metrics as the two paths provide complementary insight. The bottom of the hierarchy, common for both paths, are the factors per programming model component.

The model was initially designed for hierarchical codes where MPI is the outer level, with all ranks having the same number of threads. While evaluating the hierarchical approach, we identified how to adapt the model to be valid when there are different number of threads per MPI rank. With some further adjustments, our approach is also applicable when more than one thread of each MPI rank calls MPI or to non-hierarchical codes where the non-communicating threads are active while

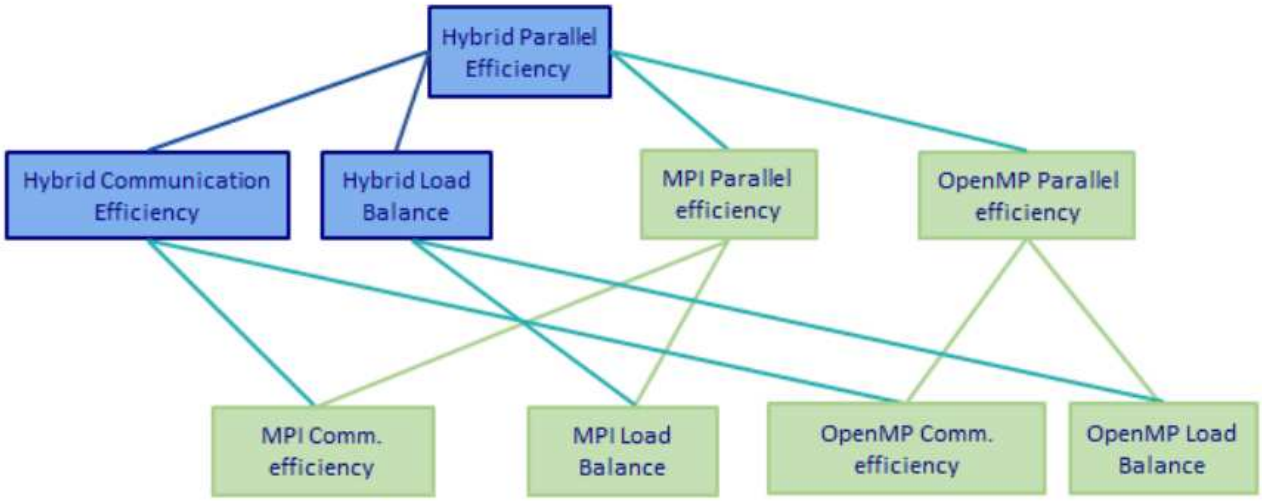


Fig. 1: Hybrid efficiency model.

the communications take place. In this section we describe the hierarchical model in detail as well as a very brief description of our plans for the non-hierarchical codes.

A. Hierarchical Codes

Hierarchical codes are the most frequent scenario for HPC codes. In many cases the initial parallelization was done with MPI and later the computation phases of the MPI code are parallelized with a second programming model. Typically, while some MPI processes are communicating, their threads are either in the idle loop (when MPI is called from a region not parallelized with OpenMP) or blocked (if it occurs within a parallel). For this scenario it is reasonable to consider that MPI is the only responsible for those time intervals, as improvements in the MPI part would also automatically reduce the threads waiting time.

To explain our approach we can consider we have a rectangular space that delimits our execution (N resources x T time) and we have to classify each sub-rectangle either as computing or inside a parallel runtime. It can be seen as the typical timeline reported by most trace-based tools. The left side of Figure 2 shows one example of activity timeline where blue corresponds to computing, red to MPI, yellow to OpenMP library and white to the idle loop.

The left side of the image corresponds to the hybrid classification where we consider both MPI and OpenMP runtimes as overheads. If we want to consider only the MPI component, we can recolor the rectangular space with a coarse granularity defined by the MPI ranks where the MPI activity delimits the MPI regions and the rest of the space is considered computation from the MPI point of view as it is displayed in the right side of Figure 2.

With the assumptions that only one thread per MPI rank communicates and that the number of OpenMP threads is the same for all the MPI ranks (in our experience this is the most frequent configuration), the same efficiencies are obtained if we only consider the MPI ranks (OpenMP master threads), ignoring the activity of all the other threads.

The equations (8), (9) and (10) compute the efficiencies for the MPI component and are based on an MPI profile applied only to the threads that communicate through MPI (in this hierarchical case, the master thread). From the point of view of an MPI profile, each rank can be either inside the MPI library, or computing (outside MPI).

$$MPI\_Par\_Eff = \frac{\sum_{(i=1)}^{P} OutsideMPI_{i,1}}{P \cdot T} \tag{8}$$

$$MPI\_Comm\_Eff = \frac{max(\{OutsideMPI_{1,1}, .., OutsideMPI_{P,1}\})}{T} \tag{9}$$

$$MPI\_LB\_Eff = \frac{\frac{\sum_{(i=1)}^{P} OutsideMPI_{i,1}}{P}}{max(\{OutsideMPI_{1,1}, .., OutsideMPI_{P,1}\})} \tag{10}$$

The metrics for the OpenMP component are computed considering that any loss of efficiency that cannot be justified by the MPI component is due to the OpenMP parallelization. This means that all the inefficiencies in the regions where there are no MPI calls have to be blamed to OpenMP. As a result, the OpenMP component efficiencies are simply the ratio between the hybrid efficiency and the MPI component efficiency (11), (12) and (13). We should remark that this approach does not require that MPI is called from outside the parallel regions.

$$OpenMP\_Par\_Eff = \frac{Hybrid\_Par\_Eff}{MPI\_Par\_Eff} \tag{11}$$

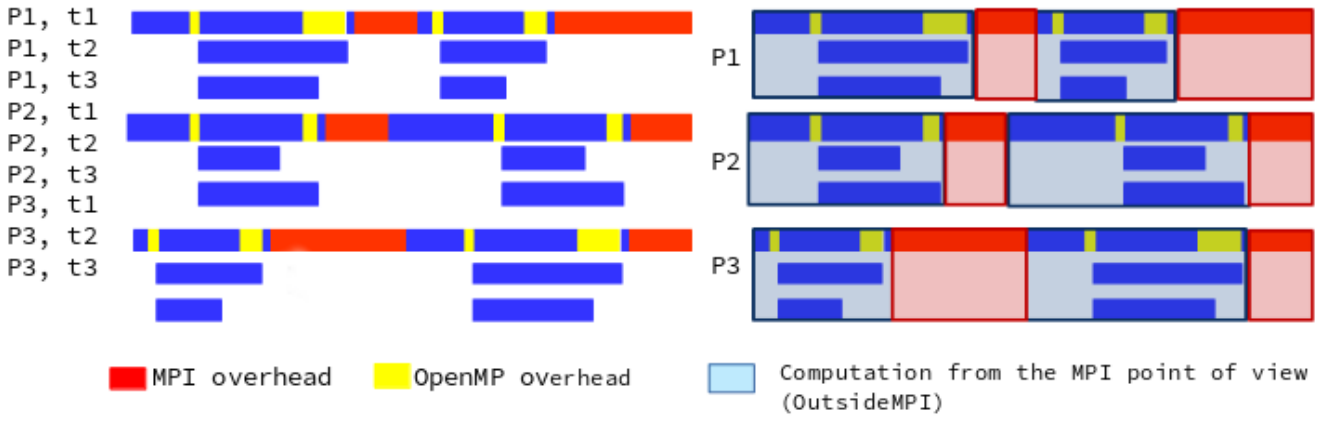


Fig. 2: Example of a hybrid timeline (left) and its MPI classification (right).

$$OpenMP\_Comm\_Eff = \frac{Hybrid\_Comm\_Eff}{MPI\_Comm\_Eff} \tag{12}$$

$$OpenMP\_LB\_Eff = \frac{Hybrid\_LB\_Eff}{MPI\_LB\_Eff} \tag{13}$$

In the infrequent case where the number of threads per MPI rank is different, the MPI efficiencies have to be rescaled taking into account the number of threads on each MPI rank (as the area each rank is responsible for would depend on its number of threads). For instance, assuming t_i is the number of threads for rank i, the MPI component efficiencies can be computed using (14), (15) and (16)

$$MPI\_Par\_Eff = \frac{\sum_{(i=1)}^{P} OutsideMPI_{i,1} \cdot t_i}{P \cdot T \cdot \sum_{(i=1)}^{P} t_i} \tag{14}$$

$$MPI\_LB\_Eff = \frac{\frac{\sum_{(i=1)}^{P} OutsideMPI_{i,1} \cdot t_i}{P \cdot \sum_{(i=1)}^{P} t_i}}{\max(\{OutsideMPI_{i,1}, ..., OutsideMPI_{P,1}\})} \tag{15}$$

$$MPI\_Comm\_Eff = \frac{\max(\{OutsideMPI_{i,1}, ..., OutsideMPI_{P,1}\})}{T} \tag{16}$$

The approach used to compute the efficiencies of the second component allows us to apply the same methodology with other programming models. Isolating first the MPI component, means blaming first to MPI, so when a given MPI rank is delayed, the corresponding waiting time is computed as MPI ineficiency despite the source of the delay may be inside OpenMP. The justification for this time distribution is that all MPI waiting time may be improved modifying the MPI part, and it is important to capture that insight.

B. Non-hierarchical Codes

There are applications with two levels of parallelism that are not hierarchical, for example, hybrid codes where OpenMP is using a task-based approach. It may be the case where MPI is called from one of the tasks that it may even be executed by a different thread on each iteration. This is also the case of a `parallel do` approach where different threads may call to MPI. Or it can be the case of an MPI+CUDA code where the communications take place overlapped with the GPU executing some kernels. In these scenarios, we have to blame MPI only for the MPI time plus the time other threads are waiting that the MPI is completed. We are currently working on the formulation as well as on identifying codes with that kind of hybrid parallelization.

IV. Experiments

In this section we describe some of the experiments carried out to validate and to explore the results of our methodology.

A. Validation

The goal of the experiments described in this section is to validate that different scenarios of load unbalance were correctly reported by the model pointing either to MPI, OpenMP or both.

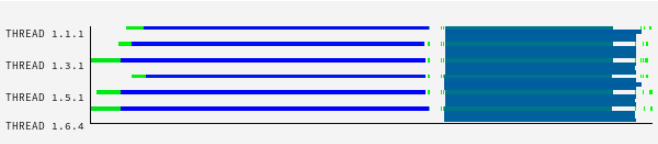
It has been done using a simple hybrid code running with 6 MPI ranks with 4 OpenMP threads each. The code at high level is:

```
Code not parallelized with OpenMP (C1)
MPI communication
OpenMP Parallel loop (C2)
MPI communication
```

Giving different weights and unbalances to the two computation phases (C1 and C2) we generated different scenarios of load unbalance symptoms. As C1 is a serial code not parallelized with OpenMP it always contributes to OpenMP efficiencies. Unbalancing C1 or C2 between MPI ranks we can

generate MPI unbalance and unbalancing C2 between threads generates OpenMP unbalance.

1) Large serial computation: For this run we configured C1 to represent close to 60% of the computing time and generated a balanced C2. Figure 3a shows the timeline of the computing phases for this configuration. The view displays the duration of the computations as a gradient (from light green low value to dark blue high value). Both C1 and C2 computations are colored in blue, while the light green regions correspond to small computations in the initialization and communication phases. The metrics reported by the proposed model are described in Table 3b. We can see that the model points to an unbalance problem in the OpenMP part that corresponds to the serial code not parallelized with OpenMP (C1).



(a) Timeline

	Hybrid	MPI	OpenMP
Parallel efficiency	47.07	95.97	49.05
Load Balance efficiency	52.09	98.87	52.69
Communication efficiency	90.37	97.06	93.11

(b) Efficiency Metrics (%)

Fig. 3: Large serial computation test.

2) Unbalanced OpenMP loop: In this case we reduced the weight of the serial computation phase (C1) to less than 15% and generated unbalance in the OpenMP parallel loop (C2) where the master thread has significantly less work. Figure 4a shows the timeline of the computing phases for this configuration. The metrics reported by the proposed model are described in Table 4b. We can see that the model points to a non-severe unbalance in the OpenMP component that measures the work distribution considering both C1 and C2. The OpenMP communication inefficiency captures the unbalance that compensates during the execution: in C1 the slave threads are waiting for the master thread and in C2 the master threads wait while the slaves are computing.

3) Unbalanced MPI + serial OpenMP: For this case we maintain the original weight of the serial computation phase (C1) and generate unbalance between the MPI ranks for the OpenMP parallel loop (C2). Figure 5 shows the timeline of the computing phases and the metrics reported by the model. We can see that the model points to unbalance problems both in MPI (due to C2) and in OpenMP (due to C1).

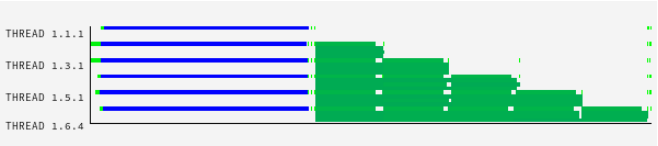
4) Unbalanced MPI: In this test we refine the previous analysis focusing only on the C2 region as an approach that can be used to identify the distribution of the unbalance between C1 and C2. Figure 6a shows the timeline of the

(a) Timeline

	Hybrid	MPI	OpenMP
Parallel efficiency	70.79	99.43	71.20
Load Balance efficiency	81.66	99.89	81.75
Communication efficiency	86.69	99.54	87.09

(b) Efficiency Metrics (%)

Fig. 4: Unbalanced OpenMP loop test.



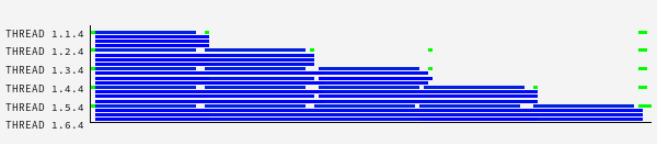
(a) Timeline

	Hybrid	MPI	OpenMP
Parallel efficiency	43.68	69.12	63.19
Load Balance efficiency	48.46	70.01	69.22
Communication efficiency	90.14	98.6	91.42

(b) Efficiency Metrics (%)

Fig. 5: Unbalanced MPI + serial OpenMP test.

computing phases for this configuration. We can see that when we apply the model only to C2 it points to unbalance in MPI, confirming the OpenMP unbalance reported in the previous test is concentrated in C1.



(a) Timeline

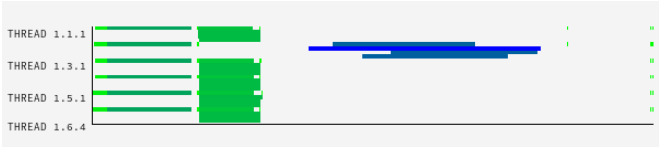
	Hybrid	MPI	OpenMP
Parallel efficiency	55.95	59.83	93.51
Load Balance efficiency	58.78	61.20	96.05
Communication efficiency	95.19	97.74	97.39

(b) Efficiency Metrics (%)

Fig. 6: Unbalanced MPI test.

5) Preempted run: This test was a non-intentional run of the large serial computation test where the execution of one of the MPI ranks was preempted. As we did not filled the node and we did not run in exclusive mode, the preemption may be caused by another job running on the same node.

We consider interesting to investigate the diagnosis for this unexpected perturbed run. Figure 7 shows the timeline of the computing phases and the metrics reported. The model points to unbalance problems in MPI (due to the delay created in MPI waiting for the preempted rank to finish) and problems of communication in OpenMP (due to the large gap in the OpenMP runtime while the rank was preempted). We consider this is the correct diagnosis as it points to the problem of the preemption in OpenMP as well as its impact in MPI.



(a) Timeline

	Hybrid	MPI	OpenMP
Parallel efficiency	18.04	38.17	47.26
Load Balance efficiency	42.48	45.54	93.28
Communication efficiency	42.48	83.81	50.69

(b) Efficiency Metrics (%)

Fig. 7: Preempted run test.

B. Lulesh analysis

One of the benefits of our model is that it can be used to analyze different hybrid parallelizations. In this section we compare the execution of Lulesh [7] running on two BSC systems: Marenostrum4 [8] and CTE-Power [9] to select the platform in which we will do the scaling test. For the first analysis we used one node on each machine (Marenostrum4 8 MPI x 6 OpenMP, CTE-Power 8 MPI x 4 CUDA devices).

The run of 100 iterations takes between 8 and 9 seconds on each machine but while the initialization in Marenostrum4 is only 90μs, initialization and finalization with CUDA is around 90% of the execution time (close to 8 seconds), so the core computation (the iterative loop) is 10x faster in CTE-Power.

Nevertheless, the efficiencies analysis using our model reports that the run in Marenostrum4 is much more efficient. While MPI+OpenMP hybrid parallel efficiency is 77%, it goes down to less than 20% in MPI+CUDA.

The MPI component parallel efficiency has closer values with 95% in MPI+OpenMP vs. 83% in MPI+CUDA. The differences in the MPI component parallel efficiency are due to a worst MPI load balance that is 88% in the execution of MPI+CUDA and goes up to 96% in the MPI+OpenMP run.

There are two reasons for the low CUDA parallel efficiency: First, the CPU spends most of the time either configuring or waiting for the device. This is common for many CUDA codes where all the computations are ported to the GPU and the CPU main role is to feed work to the accelerator. Second, the Lulesh implementation uses 10 streams per MPI rank and there is not much overlap of the kernels executed in different streams. The traces revealed that less than 10% of the time there are multiple streams running simultaneously. If we accumulate the streams for each rank, the useful time in the devices increases to 65%. Considering the accumulated metrics, the model highlights low CUDA component commmunication efficiency caused by time in the `cudaConfigureCall` and `cudaStreamSynchronize` calls. This insight suggests that we should increase the ratio between the kernel execution and the configuration and synchronization calls.

The analysis of the MPI+OpenMP configuration points to the OpenMP communication efficiency with a value of 88%. Looking at the traces we identifed a phase between the functions `CalcMonotonicQRegionForElems` and `MPI_Allreduce` where small parallel regions are repeated in a loop, which suggests to move the pragma to an outer level to reduce the overhead of the frequent fork-join synchronization.

In summary, despite the MPI+CUDA run is much faster, the MPI+OpenMP run does a better usage of the resources. Both runs report a similar behaviour with respect to MPI.

As a result of the previous analysis, we select MareNostrum4 for our scaling test. Lulesh default behavior is weak scaling, this allow us to significantly increase the scale using the same input parameters. As Lulesh requires a cube number of MPI ranks, we selected as target the cube of 6 (216). Keeping the previous configuration of 6 OpenMP per MPI rank, the scale of the run goes up to 1296 cores (previous run used 48 cores). Table I collects the nine efficiencies of the two runs to evaluate in detail the scaling behaviour.

TABLE I: Lulesh MPI+OpenMP scaling efficiencies (%)

	48(8x6)	1296(216x6)
Hybrid Parallel efficiency	77.28	64.57
Hybrid Load Balance efficiency	88.93	77.14
Hybrid Communication efficiency	86.90	83.06
MPI Parallel efficiency	95.00	84.38
MPI Load Balance efficiency	96.27	86.57
MPI Communication efficiency	98.68	97.48
OpenMP Parallel efficiency	81.35	76.52
OpenMP Load Balance efficiency	92.38	89.81
OpenMP Communication efficiency	88.06	85.21

Increasing 27x the number of cores reduces the hybrid parallel efficiency from 77% to close to 65%. The factor that reports a higher impact (reduction) at hybrid level is the load balance. With respect to the distribution between the two programming models there is a higher penalty in MPI. This is expected because we have not changed the ratio of OpenMP threads per MPI rank. What is maybe not so expected is that the degradation is related with load unbalance. MPI communication efficiency are very similar in both runs indicating there is no problem of scaling in the transfer of data. Looking at the trace we can identify that the MPI unbalance is concentrated in the two largest computation phases.

C. Application analysis

This methodology was used in the POP Audit for Tsunami-HySEA [10], the numerical model of the HySEA family de-

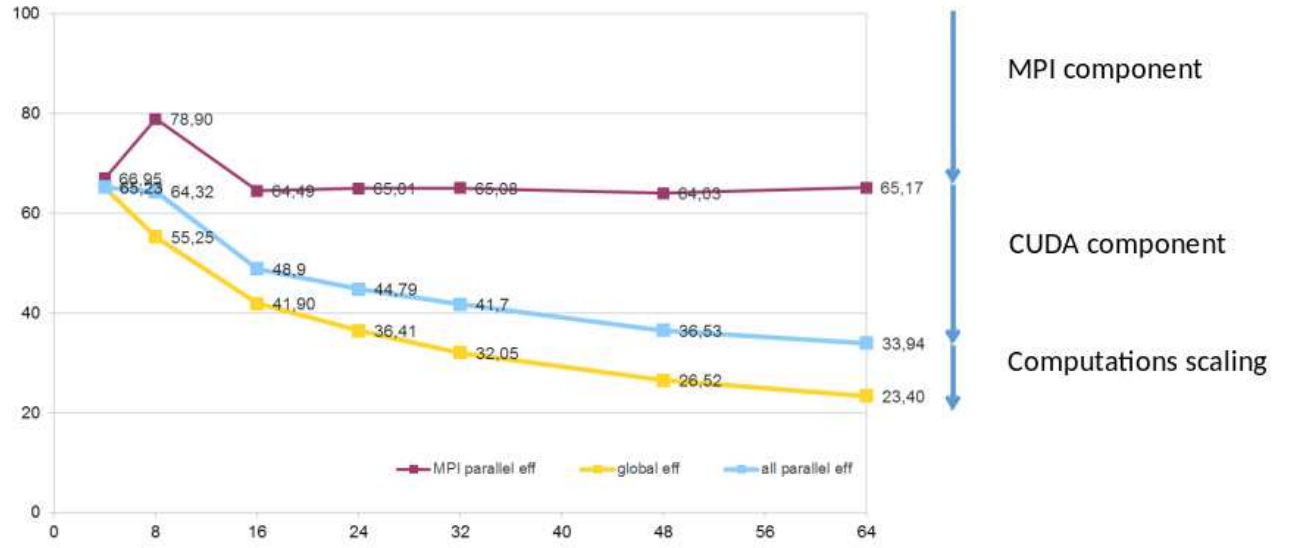


Fig. 8: Scaling of the main component efficiencies for Tsunami-HySEA.

signed for quake generated tsunami simulations. Programmed with MPI+CUDA, we ran in the BSC CTE-Power scaling from 4 CPUs + 4 GPUs to 64 CPUs + 64 GPUs using the Pacific ocean input.

We select as relevant metrics the global efficiency and the hybrid and MPI parallel efficiencies, plotting their scaling values in Figure 8. The gap between the top of the image and the MPI parallel efficiency corresponds to the loss of efficiency caused by MPI, the area between the two parallel efficiencies is the impact of the CUDA component and the region between the hybrid parallel efficiency and the global efficiency indicates the efficiency scaling the computations.

Looking at this plot we can say that MPI is the programming model with a higher impact on the loss of efficiency (close to 35%) despite their contribution to the global parallel efficiency does not degrade with the scale drawing almost an horizontal line except for 8 CPUS + 8 GPUs.

The GPU part shows a lower impact but increasing with the scale, being closer to the MPI contribution on the largest configuration. Finally, the computations have the smaller contribution despite they also reflect a small degradation with the scale. So, the higher loss of efficiency is on the MPI layer, but the scaling problems are related to the CUDA parallelization and with a very small impact, the scaling of the computations.

The analysis of the MPI parallel efficiency components indicated that the main source of loss of efficiency is load balance (with values around 75%). Analysing the unbalance in the traces, we can see that it is a structured unbalance that is maintained in all the iterations, suggesting it is due to the work distribution, and that improving the initial distribution would improve the execution of all the iterations.

The analysis of the GPU component identifies that the main problem is due to the communications, and the traces revealed it is caused by the increase on the `cudaMemcopy` runtime call used to copy data from and to the device. The `cudaMemcopy` call represents 24% of the execution time on the CPUs with the largest configuration. The GPU kernels show a reasonable scaling between 10x to 14x with 16x more resources, being this small reduction of the scaling the source of degradation detected in the computation scalability.

The feedback provided by this analysis was considered very useful by the code owner that is currently working on a new improved version.

V. Related Work

Several solutions have been proposed to model the behavior of parallel applications and measure the efficiency. The most common approach is to build specific models for a particular paradigm. Sun et al. [11] introduced a method to predict the performance of MPI programs. Based on random forest machine learning they predict the performance by considering a number of factors such as values of variables, counters of branches, loops and MPI runtime features related to the data size and the number of targets of MPI calls. Similarly, we rely on MPI instrumentation, but our model focuses on the percentage of execution time of the computation phases delimited by calls to the MPI runtime.

Dietrich et al. [12] defines common inefficiency patterns for computation offloading models such as CUDA, OpenCL, OpenACC, and OpenMP target. By using pattern analysis techniques they identify that the most common offloading inefficiencies are the early wait-for-device operations and the device idle time. While these metrics focus on particular sources of offloading problems, our model provides more general efficiency metrics that describes common problems for different paradigms, and more importantly, are reported for each component of a hybrid parallelization.

Targeting hybrid models, Wu and Taylor [13] present a performance modeling framework based on memory bandwidth contention time and a parameterized communication model to predict the performance of OpenMP, MPI and hybrid applications. They propose an additive hybrid model for MPI+OpenMP applications. In contrast, our hybrid model is multiplicative for MPI+X applications, where *X* might be different paradigms such as GPU, OpenMP and others.

Other authors use empirical approaches not focusing on the parallel paradigm. Calotoiu et al. [14] proposed a method to generate empirical scaling models from a limited set of performance measurements as a function of an arbitrary set of input parameters. This approach builds models tailored to each target application based on specific parameters, and requires multiple experiments to consider all combinations of parameters. The PMaC tool suite creates scaling models of parallel applications based on modeling single-processor performance considering features related to processor and network architecture, and was extended to model accelerators [15]. Goldsmith et al. [16] describe the behavior of programs by measuring their empirical computational complexity, fitting a model that predicts performance as a function of workload size. While the previous works explored the direction of performance modeling or prediction based on specific applications and system characteristics, our approach aims at modeling and decomposing the efficiency of an application into a small set of metrics that reflect the common causes of inefficiency for each component of hybrid parallel programs. This is extremely useful to characterize the performance behavior of an application at different stages of optimization and tuning, as well as to easily compare performance across different processes, machines, or execution scenarios.

As a summary, the main novelty of our work is to provide a generic multiplicative model to measure the efficiency of hybrid applications' executions.

VI. Conclusions and Future Work

In this paper we present a model to characterize the efficiency of hybrid codes, being able to determine the contribution of each programming model. The model is based on fundamental factors inherent to all parallel programing models: time outside the parallel libraries (parallel efficiency), work distribution (load balance) and synchronization between processes/threads that is not caused by the global unbalance (communication efficiency).

These fundamental factors are intuitively identified in a message passing approach like MPI, nevertheless the metrics can be mapped into any parallel paradigm, for instance OpenMP communication efficiency identifies both temporal unbalances that are compensated along time as well as time on the OpenMP library caused by locks or due to the selected scheduling. We base our approach on what programming models have in common, instead of focusing on their specificities.

Despite the model was initially developed targeting hierarchical hybrid codes, the same approach can be used when there is more overlap between the two programming models. We are currently working on the required extensions to support them.

VII. Acknowledgments

This work has been partially developed under the scope of POP CoE which has received funding from the European Union´s Horizon 2020 research and innovation programme (under grant agreements No. 676553 and 824080), and with the support of the Comision Interministerial de Ciencia y Tecnología (CICYT) under contract No. PID2019-107255GB-C22. We also want to acknowledge the ChEESE CoE and the EDANYA group from Universidad de Málaga (www.uma.es/edanya) that granted us permission to report on the Tsunami-HySEA analysis.

References

[1] Rosas, C., Gimnez, J., Labarta, J.: Scalability prediction for fundamental performance factors. Supercomputing Frontiers and Innovations (2014), https://superfri.org/superfri/article/view/7

[2] POP: POP Center of Excellence, https://pop-coe.eu

[3] BSC Performance Tools, https://tools.bsc.es

[4] Casas, M., Badia, R., Labarta, J.: Automatic analysis of speedup of mpi applications. In: Proceedings of the 22Nd Annual International Conference on Supercomputing. pp. 349–358. ICS '08, ACM (2008), http://doi.acm.org/10.1145/1375527.1375578

[5] Dimemas, https://tools.bsc.es/dimemas

[6] Wagner, M., Mohr, S., Giménez, J., Labarta, J.: A Structured Approach to Performance Analysis. In: Niethammer, C., Resch, M.M., Nagel, W.E., Brunst, H., Mix, H. (eds.) Tools for High Performance Computing 2017. pp. 1–15. Springer International Publishing, Cham (2019)

[7] Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Tech. Rep. LLNL-TR-641973 (August 2013)

[8] MareNostrum 4 overview. https://www.bsc.es/support/MareNostrum4-ug.pdf

[9] CTE-Power overview. https://www.bsc.es/support/POWER_CTE-ug.pdf

[10] Macas, J., Castro, M., Ortega, S., Escalante, C., Gonzlez-Vida, J.: Performance Benchmarking of Tsunami-HySEA Model for NTHMPs Inundation Mapping Activities. Pure and Applied Geophysics **1**(37) (2017), https://doi.org/10.1007/s00024-017-1583-1

[11] Sun, J., Zhan, S., Sun, G., Chen, Y.: Automated performance modeling based on runtime feature detection and machine learning. In: 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC). pp. 744–751 (Dec 2017). 10.1109/ISPA/IUCC.2017.00115

[12] Dietrich, R., Tschüter, R., Juckeland, G., Knüpfer, A.: Analyzing offloading inefficiencies in scalable heterogeneous applications. In: Kunkel, J.M., Yokota, R., Taufer, M., Shalf, J. (eds.) High Performance Computing. pp. 457–476. Springer International Publishing, Cham (2017)

[13] Wu, X., Taylor, V.: Performance modeling of hybrid mpi/openmp scientific applications on large-scale multicore supercomputers. J. Comput. Syst. Sci. pp. 1256–1268 (2013), http://dx.doi.org/10.1016/j.jcss.2013.02.005

[14] Calotoiu, A., Beckinsale, D., Earl, C.W., Hoefler, T., Karlin, I., Schulz, M., Wolf, F.: Fast multi-parameter performance modeling. In: 2016 IEEE International Conference on Cluster Computing (CLUSTER). pp. 172–181 (Sep 2016). 10.1109/CLUSTER.2016.57

[15] Meswani, M.R., Carrington, L., Unat, D., Snavely, A., Baden, S., Poole, S.: Modeling and predicting performance of high performance computing applications on hardware accelerators. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum. pp. 1828–1837 (May 2012). 10.1109/IPDPSW.2012.226

[16] Goldsmith, S.F., Aiken, A.S., Wilkerson, D.S.: Measuring empirical computational complexity. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. pp. 395–404. ESEC-FSE '07, ACM (2007), http://doi.acm.org/10.1145/1287624.1287681