Towards a Better Expressiveness of the Speedup Metric in MPI Context

Jean-Baptiste Besnard
ParaTools SAS
Bruyères-le-Châtel, France
Email: jbbesnard@paratools.fr

Allen D. Malony
ParaTools Inc.
Eugene, OR, USA
Email: malony@paratools.com

Sameer Shende
ParaTools Inc.
Eugene, OR, USA
Email: sameer@paratools.com

Marc Pérache
CEA, DAM, DIF
F-91297 Arpajon, France
Email: marc.perache@cea.fr

Patrick Carribault
CEA, DAM, DIF
F-91297 Arpajon, France
Email: patrick.carribault@cea.fr

Julien Jaeger
CEA, DAM, DIF
F-91297 Arpajon, France
Email: julien.jaeger@cea.fr

***Abstract*—Many-core processors are imposing new constraints to parallel applications. In particular, the MPI+X model or hybridization is becoming a compulsory avenue to extract performance by mitigating both memory and communication overhead. In this context, performance tools also have to evolve in order to represent more complex states combining multiple runtimes and programming models. In this paper, we propose to start from a well-known performance metric, the Speedup, showing that it can be bounded by the acceleration of any program section. From this observation, we propose a compact tool-oriented MPI abstraction providing such time slices (or phases). We demonstrate the benefits of this approach first on a simple benchmark, identifying factors limiting speedup. And second, using an MPI+OpenMP benchmark to measure OpenMP scaling solely from MPI instrumentation.**

1. Introduction

As supercomputer architectures evolve towards many-core processors able to run hundreds of threads in shared-memory, MPI-only applications have to cope with increasing constraints. In particular, domain replication will incur more memory and communication pressure as concurrency increases. Applications are then bound to evolve towards the MPI+X model, mixing a distributed programming model with shared-memory multi-threading. From a performance tool point of view, this hybridization increases the number of parameters impacting program efficiency and consequently tool complexity and overhead.

In this paper, we consider the familiar *Speedup* metric in the new reality of hybridized MPI applications. Our point is to derive and discuss potential methods and tools which would help to better understand the behavior of a parallel application when scaling. Indeed, when considering simulations that split a data domain over several distributed computation units, a common pattern in HPC, the question of their scalability on an increasing number of cores arises naturally in the light of the many-core transition. A common way of measuring the effective scaling of an application is the Speedup (S) metric which defines the acceleration provided by p processing units working in parallel, $par(n, p)$, relative to a sequential execution $seq(n)$ with n denoting the problem size. This yields the Speedup equation:

$$S(n,p) = \frac{seq(n)}{par(n,p)} \tag{1}$$

This equation is at the heart of HPC workloads as it determines if a given problem can either be processed faster, *strong scaling*, or if a larger workload can be processed in the same time, *weak scaling*, taking advantage of more computation units. The object of this paper is to discuss the parameters driving scalability in a practical manner and to develop a very compact MPI abstraction aimed at outlining phases in MPI applications in a tool-independent manner.

First, we will justify why analyzing parallel program sections as "time slices" is of interest to derive speedup boundaries. From this observation, we propose the *partial speedup bounding* as a practical scalability law.

We argue that there are generally other sources of overhead beyond communications that limit performance scaling. The problem is how to provide a uniform means to capture the context of these overheads for the *partial Speedup bounding* approach. We propose a simple `MPI_Section` labeling interface that allows tools to take advantage of the MPI profiling support to obtain context and performance data. The proposition is discussed first through a scaling analysis of a convolution benchmark which (as in many HPC applications) presents code sections with varying scalability that would gain to be analyzed separately. This is possible with the extra semantics provided by the `MPI_Section` primitive. A second analysis with the Lulesh MPI+OpenMP CORAL benchmark demonstrates the use of MPI Section for OpenMP characterization, this without requiring any OpenMP instrumentation.

The paper makes several contributions, including the partial Speedup bounding methodology, the MPI Section abstraction for section-based speedup analysis, a reference

1530-2016/17 $31.00 © 2017 IEEE
DOI 10.1109/ICPPW.2017.45

implementation of `MPI_Section`, and demonstration of the approach on MPI+X benchmarks. We also compare our work with prior research and outline future directions.

2. Partial Speedup Bounding

Speedup is a fundamental concept in parallel scaling laws for HPC and has been intensively modeled and discussed [1], [2], [3], [4], [5]. Our interest in this paper is to build on seminal works and derive a practical expression of scalability bounds which describes the performance of actual applications. In practical terms, we are concerned with the case where a user observes that his HPC simulation program fails to scale and wants to understand why. We will first start from common scaling boundaries which are all derivative of the Speedup equation we presented in introduction of this paper. Then, we will present our *partial Speedup* equation and the *partial Speedup bounding* laws it supposes. From this model we argue that the overall scalability of a parallel application as universally measured by HPC programmers could be easily improved to allow a better understanding of parallel scaling, particularly as applications are gaining in heterogeneity due to the upcoming many-core context.

$$S(n,p) \leq \frac{1}{f_s + \frac{f_p}{p}} \leq \frac{1}{f_s} \tag{2}$$

Amdahl's law [6] (Equation 2) is directly derivable from the canonical Speedup equation with f_s and f_p representing the sequential and parallel fractions of the code, respectively ($f_s + f_p = 1$). It is a well known speedup bound particularly due to its dramatic conclusion for large p (i.e., speedup is asymptotically bound by the sequential fraction of the program). Note that this is true in the "strong scaling" configuration and therefore with a fixed problem size. This law can easily be defined as the expression of the convergence towards the non-parallelized time when dividing the parallel section.

In complement, the Speedup formulation represented by the Gustafson-Barsis [7] scaled Speedup or the Karp-Flatt Metric [8] argues that an increasing number of resource is generally associated with an increasing problem size In practice, simulation applications are between these two configurations, sometimes willing to capture more of the simulated phenomenon and some other times trying to reduce the time to result for a given problem. This creates a spectrum of strong and weak scaling scenarios, posing problems of interpretation of the Speedup metric which dramatically varies, particularly in function of problem size.

$$S(n,p) \leq \frac{\sum_i T_{s_i}}{\sum_i T_{p_i}} \tag{3}$$

As presented in Equation 3, we propose to model the application as a sum of contributing times, both in the parallel and in the sequential configurations, respectively represented by T_{p_i} and T_{s_i}. Each of these contribution being related to distinct program sections (indexed by i). Moreover, we assume that each of these quantities are function of both n the problem size and p the number of computational units such as $T_i = f_i(n,p)$, an arbitrary positive function. Moreover, by definition of the sequential time, Speedup equation numerator is such as $p = 1$ – running on a single computational unit, yielding the following Speedup model:

$$S(n,p) \leq \frac{\sum_i f_i(n,1)}{\sum_i f_i(n,p)} \tag{4}$$

From this simple formula, we can see that the sequential time is modeled with the same functions as the parallel time, speedup being the acceleration achieved in function of p. Here, we clearly see that the contribution of the n parameter is a function of the *kind* of speedup, weak or strong scaling. When considering strong scaling, $n = n_0$, a fixed value determining the size of the sequential problem. This immediately leads to a constant numerator and to a denominator which is only a function of p such as:

$$S_{strong}(n,p) \leq \frac{\sum_i f_i(n_0,1)}{\sum_i f_i(n_0,p)} \tag{5}$$

Here, we are then analyzing the f_i in function of p, and one can immediately see that each of these factors is by itself bounding the overall speedup, the denominator being a sum of positive terms. In other words, we can bound the strong scaling Speedup as follows, a process that we call *partial Speedup bounding*:

$$\forall i, S_{strong}(n,p) \leq \frac{\sum_i f_i(n_0,1)}{f_i(n_0,p)} \tag{6}$$

This bound simply expresses the fact that any subpart of the application which does not scale properly is by itself putting an upper bound on the strong scaling Speedup. From a more empirical point of view, we can see the numerator as a parallel budget, where some processing are easily parallelized, their total time decreasing functions of p, up to a point where less optimized factors prevent further scaling. Using this formula, we can show that if a Section stops accelerating at a given p, called inflexion point later in the paper, it is already putting an upper bound on the overall speedup whereas other boundaries such as Amdahl's only consider the limit when $p \to +\inf$. Moreover, this formula is directly connected to measurable section timing unlike Amdahl's law which requires the definition of a sequential fraction which is not easily correlated to actual code regions – sequential fraction being generally measured in practice through speedup limit.

Thus, starting from this very practical speedup bound, we present a possible MPI-based method to outline various parts of a given code execution, thereby allowing tools to perform scalability analysis on individual code sections in a more efficient manner. Currently, profiling tools must infer the internal semantics of a section. For example, in Figure 4, we outlined several parts of the execution involving various low-level operations, such as IO or communications. However, from a profiling tool point of view, extracting such macroscopic information can be difficult as the state can be diluted in overlapping code regions. A tool could profile exclusive time in each function and perform a scalability analysis on these durations, but it would not be able to extract in a straightforward manner distributed computation phases.

Our proposal for `MPI_Sections` below will enable any MPI programmer to inform profiling tools of such distributed code region with minimum code modification. We believe that the integration of such standardized mechanism in MPI applications would provide valuable information to support tools, opening the way to interesting analysis relatively to load-balancing and of course scalability.

We have shown that scalability can be impacted by multiple factors, we are now going to consider the most common one, communications. One interesting aspect of communications is that their sequential time is null, creating a pure overhead. We will show that the mitigation of this overhead factor in a many-core context, with memory constraints advocates for an MPI+X approach, combining shared and distributed memory parallelism.

3. Distributed-Memory Constraints

It is now a common observation that distributed-memory programming relying on message passing models implemented in MPI are challenged by the new many-core architectures. This evolution in the execution substrate requires modifications inside parallel codes to harness the power of these new architectures. With modern many-core devices and their support for hardware threading (e.g., the Intel KNL has 68 cores each with 4 hyper-threads), the memory per MPI rank on these processor is decreasing.

Distributed-memory programming models have been developed to exchange data between memories. This directly leads to both memory duplication (saving a remote state) and communication overhead. limit communication overhead. For stencil-based simulations, it is known that the halo-cells ratio directly linked with communication size is smaller for large memory areas [9]. Unfortunately, higher dimension domain decompositions require larger local domains to minimize this memory overhead. As a consequence, a simulation program relying on a 3D communication scheme requires larger memory areas to mitigate its communication overhead.

When developers consider running simulations on larger machines with great concurrency, these constraints will be more severe and will lead to scaling inefficiencies. The conclusion is that the MPI model alone is not able to efficiently run on new many-core architectures and model mixing is becoming compulsory. MPI + X or hybrid programming models addressing larger memory regions are required. By mixing a shared-memory programming model (OpenMP) and a distributed memory one (MPI), the memory and parallel overhead constraints can be mitigated. However, hybridization introduces difficulties in the measurement and analysis of performance due to the fact that tools are now required to measure two different programming models and combine the results. These models are operating at various scales and rely on different interfaces (e.g., MPI + OpenMP, MPI + CUDA, MPI + Pthread). One observation is that MPI is often a common denominator between applications, being the de-facto distributed-memory programming model. In the following Sections, we propose to provide a simple solution to this hybrid instrumentation problem by providing a new semantic relying the common denominator that is MPI. In particular, we will show how MPI centric measurements as `MPI_Sections` achieve to capture the OpenMP scaling and its influence on the overall speedup.

4. MPI_Section Abstraction

So far, we described how the Speedup metric can be expressed from a linear combination of individual code sections running in parallel. We also presented the *partial Speedup boundary*, a simple model that captures how any poorly parallelized code section can prevent the strong scaling of an application. From this observation, we discussed the need to observe macroscopic code regions and measure the execution of phases, in order to identify and characterizes poor scaling regions in the program. This will complement the time-based and resource-based data that parallel performance tools currently provide.

Then, we discussed distributed-memory limitations, clearly demonstrating that MPI applications will have to rely on a combination of distributed and shared memory programming in order to control memory constraints at higher problem dimensions, brought on by halo-cells and transitive communication overhead. More generally, it is expected that parallel programs will be less regular [10], involving multiple level of parallelism interleaved with communication phases [9]. This poses the question of how to measure scalability when acceleration spans multiple programming models (MPI+X) or scales. How can a performance tool semantically outline parts of the execution in such hybrid context.

This section proposes a compact MPI-level interface, `MPI_Section`, based on the concept of a *section*, defined as a temporal outline of a distributed code region entered by all the MPI Processes belonging to a given communicator. Entering an `MPI_Section` is a non-blocking collective call. The idea is that a user should be able to semantically describe regions of the program with minimal code addition.

```c
/* Enter an MPI_Section */
int MPIX_Section_enter( MPI_Comm comm,
                        const char * label );
/* Leave an MPI_Section */
int MPIX_Section_exit( MPI_Comm comm,
                       const char * label );
```

Figure 1. The MPI Section interface

The two calls shown in Figure 1 are asynchronous collective calls on the target communicator, defining respectively the entry and exit point of each `MPI_Section`. A `MPI_MAIN` section is entered in `MPI_Init` and left in `MPI_Finalize`. Sections can be stacked, meaning that a section possibly contains several sub-sections. However,

```
/* An MPI_Section was entered */
int MPIX_Section_enter_cb( MPI_Comm comm,
                           const char * label,
                           char data[32]);
/* An MPI_Section was left */
int MPIX_Section_leave_cb( MPI_Comm comm,
                           const char * label,
                           char data[32]);
```

Figure 2. The MPI_Section callback interface

sections are always perfectly nested, entered in the same order and exited in the opposite order. Nested sections allow phase performance to be observed at various granularities. Also, the `MPI_Section` primitives can be used to outline shared-memory phases, as long as they are entered by all members of the communicator.

Thanks to this very compact interface, application programmers are only required to manipulate two function calls to provide additional information relatively to code semantics. Moreover, MPI_Section's semantic goes beyond simple durations. Indeed, a function call only quantifies a sequential time whereas a Sections describes a parallel slice of the execution compatible with Speedup equations covered at the beginning of this article. In other words, and as we will further elaborate on, a poorly scaling MPI_Section is directly putting an upper-bound on the speedup as shown in Equation 6.

On runtime side, the invariants relatively to section entry have to be verified using non-intrusive synchronization primitives which could for example be selectively enabled to minimize section impact. This check is important to allow profiling tools to do assumptions relatively to sections' data, for example, computing exclusive and inclusive times. As presented in Figure 2, tools are notified of `MPI_Section` events via two MPI functions which are to be intercepted through the PMPI profiling interface (their PMPI version being possibly empty if the runtime ignores such events). A profiling tool redefining those functions is able to intercept Section events in a straightforward manner. Moreover, this notification mechanism is also relatively easy to implement inside MPI runtimes. Our reference implementation simply manipulates a stack of contexts for each communicator, calling tool callbacks upon *enter* and *exit* events. Note that a 32 bytes *data* argument is also included in order to allow tools to provide additional context, for example, its own synchronized time-stamps. This argument being preserved by the MPI runtime between enter and leave events.

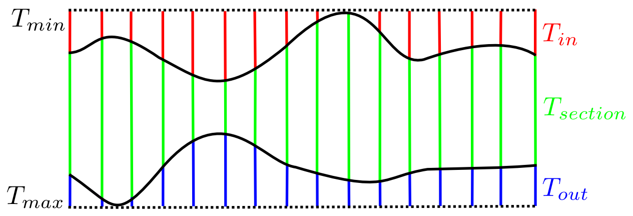


Figure 3. Illustration of the temporal layout of an MPI_Section with associated derived metrics.

One of the main motivations behind MPI_Sections is to incite the user to use them without having to link to a particular tool, this would then act as a common interface freely usable by profiling and debugging tools thanks to MPI's wide availability. As presented in Figure 3, `MPI_Sections` would provide tools with a wide range of informations for a minimal user effort:

- T_{min} is the time at which the first process enters the region of interest (outlined in green);
- T_{in} are the respective timestamps on each rank at the point of entering the region;
- T_{out} is the timestamp at which each process leaves the section;
- $T_{section}$ is the time spent in the section by each process, it is equal to $T_{out} - T_{min}$;
- T_{max} is the timestamp at which the last leaves the section.

Looking at the metrics derived from Figure 4, it can be seen that a Section can account for more than the time spent in a given region, which is usually presented in average as the sum of individual $T_{section}$ times divided by p. Indeed, phase outlining also accounts for performance variability, describing how a region was distributively entered. For example, the average and variance of the entry imbalance defined for each MPI process as $imb_{in}(p) = T_{in} - T_{min}$ can provide as compact values a representative idea of how a given code Section is entered. For example, loosely synchronized MPI ranks may avoid an `MPI_Barrier` call which would convert the imbalance in a parallel synchronization cost. Similarly, the average imbalance of a Section can be defined as $imb = (T_{max} - T_{min}) - \overline{T_{section}}$, describing in a single number the level of imbalance for a given program region. These few examples show that despite its simplicity, the `MPI_Section` approach can be used to provide interesting metrics due to its distributed nature. Metrics that we will leverage in the following section to demonstrate scalability issues through an early MPI_Section implementation.

5. Benchmarking MPI_Sections

In this section we apply an initial `MPI_Section` measurement interface to two different benchmarks in order to derive their respective scalability behavior. We will begin with a simple convolution benchmark to introduce the model. Then, we will extend this model to a more complex MPI+X application, the Lulesh Corals Benchmark.

5.1. MPI_Section on the Convolution Benchmark

This section introduces a simple benchmark to model an HPC application relying on regular domain decomposition. Such applications are very common among simulation codes, and are a good example of the applications with phase-based execution behavior. Clearly, there are a variety of computational models used in HPC applications, resulting in different execution behaviors and communication patterns. Performance tools generally are not privy to the computational models and must extract behavioral information from execution measurement to understand performance characteristics. In order to precisely control the computational model and execution parameters, we decided to write our own benchmark for demonstration purposes. We chose an image convolution problem for both its simplicity and proximity with other algorithms (e.g., Lattice-Boltzmann) where spatial values are propagated using similar stencils. To model the temporal behavior of a simulation code as a sequence of phase-based time steps, we ran a large number of convolutions, each of them involving halo-cell exchanges followed by computation.

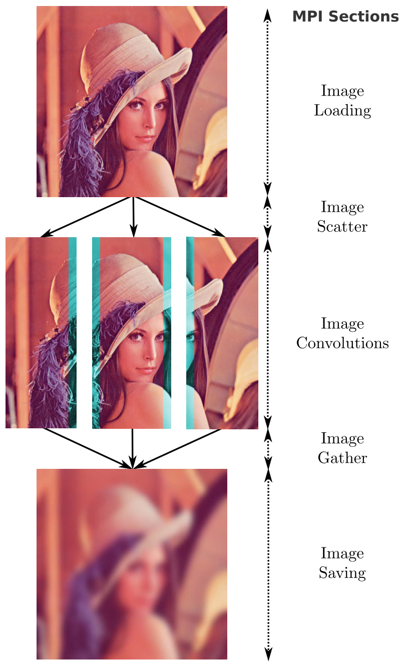


Figure 4. Schematic view of the steps involved in our simple convolution benchmark, including initialization and finalization.

As illustrated in Figure 4, our benchmark consists in applying a mean filter several times to a three-channel RGB image stored in double precision (our actual test image size is 5616×3744). The image is first loaded by `rank 0` before being scattered through a 1D splitting among the MPI processes. Then a repetitive convolution benchmark is applied to the image exchanging ghost-cells at each time-steps to satisfy spatial dependencies of the stencil. Then each subpart of the image is gathered in `rank 0` before being stored in the file-system as a result of the computation. Our benchmark also has the advantage of accounting for these extra setup and tear-down times which are needed in any parallel application, loading a data-set from a persistent storage. This benchmark has been instrumented with `MPI_Sections` describing all execution phases (see Figure 4), including halo-cells exchange as follows:

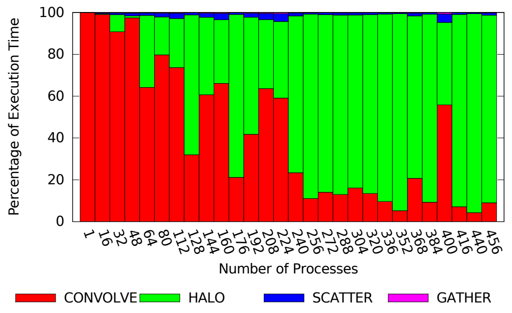
- LOAD : time to load and decode the image (sequential on rank 0), other processes waiting;
- SCATTER : scattering of image data to separate MPI processes (MPI_Scatter);
- CONVOLVE : convolution of the image region on each MPI process;
- HALO : halo-cell exchange at each convolution step between MPI processes;
- GATHER : time to retrieve the data from remote processes (MPI_Gather);
- STORE : time to store and encode the image (sequential on rank 0), other processes waiting.

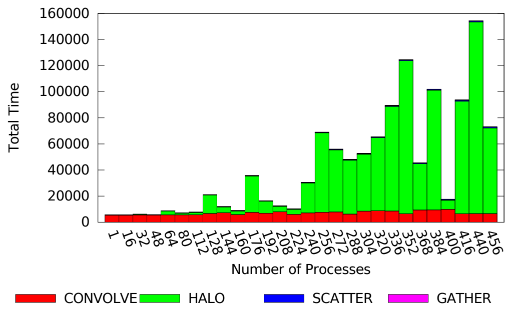
In order to perform following measurements, we ran our convolution benchmark, convolving 1000 times our reference image on an Intel Nehalem test cluster up to 456 cores. Each node consists of a single eight core Intel Xeon X5560 processor with hyper-threading disabled and 24 GB of memory. Runs were done twenty times and averaged.

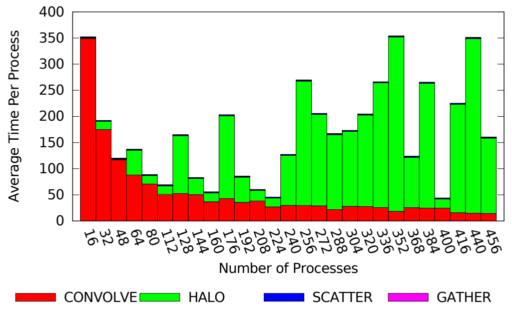
As presented in Figure 5, we can see that `MPI_Section` based profiling can be used to characterize the Speedup of an application. First, in Figure 5(d), we can see that the Speedup of our benchmark is rapidly bounded in the 64 processes range, before becoming relatively subject to noise (despite we present timing results averaged twenty times). If we now take into account data retrieved thanks to `MPI_Section` labeling, we have a better understanding of the limiting factors. Figure 5(a) presents the percentage of the execution spent in each section, we can see that the convolution time which was dominating sequentially rapidly decreased to be replaced by communication overhead.

What can be surprising here is that when splitting in 1D as done is this benchmark, the number of halo-cells is constant as is communication size per process – contradicting measurements. Figure 5(b), shows that overall communication time is an increasing function of the number of processes, it also varies a lot, we suppose that this variation comes from the decreasing computation time which does not recover communication jitter, leading to an accumulation of this variability when doing the 1000 time-steps. Figure 5(c) shows the average time per process spent in each section, it outlines the speedup achieved when increasing the number of processes and the associated communication overhead, becoming the main speedup bound. Wall-time decrease linked with communication time irregularity impacts the Speedup metric as shown in Figure 5(d). In these over-scaled configurations, measurement noise was more important due to the decreasing walltime. Despite the averaging this led to non-negligible variations as observed for 400 MPI processes in Figure 5(a).

Looking at Figure 5(b), it is relatively easy to observe the point where communication time starts to be non-scalable. Indeed, starting at 64 MPI processes (8 nodes), the computation time decreases, combined with the increasing communication cost. At larger scales, the communication has an increasing trend with non-negligible noise. If we consider the partial Speedup bounding approach and data obtained through MPI_Sections we can infer a Speedup bound $B(p)$ from the communication cost, as done in Figure 6. This


(a) Percentage of execution time per MPI_Section


(b) Total time per MPI_Section


(c) Average time per process for each MPI_Section. The sequential case with 5589,84 seconds in total was ommited for readability.

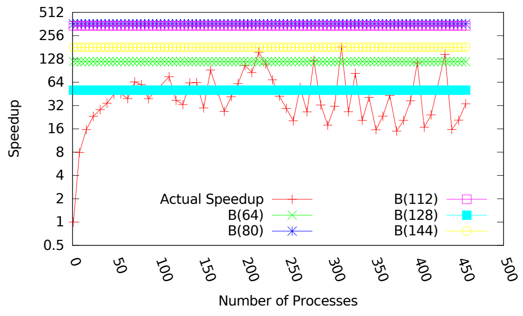

(d) Average Speedup and some of the predicted partial speedup boundaries (B) for the HALO section.

Figure 5. Execution scattering for our convolution benchmark outlined with MPI_Sections. File loading and saving were omitted due to negligible contributions.

#Processes	Tot. HALO Time	Speedup Bound (B)
64	3025.44	118.25
80	1288.64	363.96
112	1822.38	343.54
128	14135.56	50.61
144	2716.03	181.17

Figure 6. Inferred partial Speedup boundaries from ghost-cell exchange time (HALO section) on the convolution benchmark.

boundary is computed following Equation 6 considering the average ghost-cell exchange time (HALO). For example, with 64 processes, $B(64) = \frac{5589{,}84}{\frac{3025.44}{64}} = 118,25$ with 5589.84 seconds being the total section time for the sequential program. We now consider the transposition of partial speedup boundaries from Figure 6 to the actual measured speedup of Figure 5(d), it can be observed that boundaries obtained at lower scales did provide representative Speedup bounds at higher scales, this despite the important noise on the HALO MPI_Section.

5.2. MPI_Section on Lulesh

In this Section we investigate the use of MPI_Sections in the context of a real benchmark. We retained the Lulesh MPI+OpenMP code which is part of the Corals benchmark suite. We added 21 sections in the main source file in order to outline main computation steps. Then we ran Lulesh with 110 592 elements in various MPI+OpenMP configurations in order to assess its scalability on both an Intel KNL (68 cores with 4 hyper-threads) and a dual Intel Broadwell machine (2 sockets with 18 cores with two hyper-threads).

#MPI Processes	Lulesh size (-s)	Number of elements
1	48	110 592
8	24	110 592
27	16	110 592
64	12	110 592

Figure 7. Strong-scaling configurations used for Lulesh

As shown in Table 7, Lulesh requires the number of MPI processes to be a cube, however it does not put constraints on the number of OpenMP threads. Moreover, as shown in these configurations, we configured Lulesh so that the overall number of elements is constant in all configurations. This is a deliberate choice to study the strong scaling behavior of the code. Indeed, this configuration outlines more rapidly scaling issues, unlike Lulesh default behavior which scales problem size with the number of MPI processes. These parameters being fixed, we ran Lulesh on the two target machines in all configurations mixing MPI and OpenMP.

From our measurements, we observed that the time-loop section was accounting for 99% of the main function time. And within this section, two portions of the code were contributing to most of the time, `LagrangeNodal` and `LagrangeElements`. In Figure 8, we present the

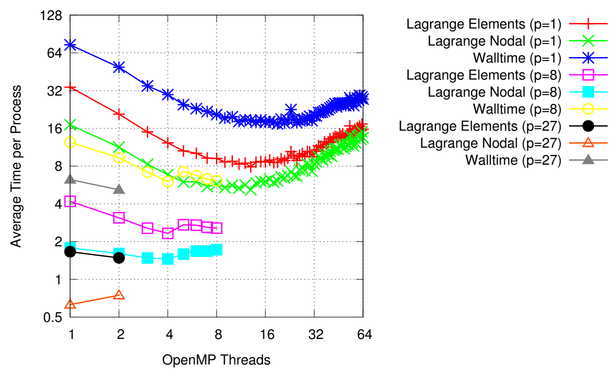


Figure 8. Lulesh MPI_Sections on a dual Broadwell machine in various MPI+OpenMP configurations. p is the number of MPI Processes.

scalability of these two sections with are mutually exclusive and which contribute to most of the the main section (denoted walltime). From this simple representation directly extracted from sessions, we can see that in this strong-scaling configuration, MPI is providing more acceleration than OpenMP. Nonetheless, OpenMP is advantageous when the problem is large as for example when running on a single process. For this particular machine it is then more optimal to parallelize on top of MPI, OpenMP providing some additional acceleration when the problem is sufficiently large.

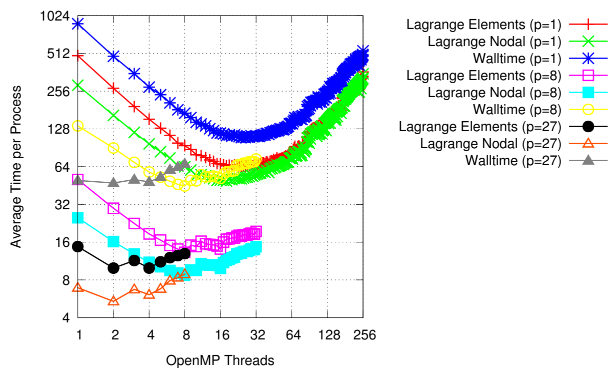


Figure 9. Lulesh MPI_Sections on an Intel KNL in various MPI+OpenMP configurations. p is the number of MPI Processes.

In Figure 9, we present a scaling analysis on an Intel KNL. Results are comparable to the ones obtained on the dual Broadwell, `LagrangeElements` providing most of the acceleration in OpenMP. There are however two differences. First, the OpenMP overhead tends to increase more rapidly than on the Broadwell. Second, in the MPI cases at 27 and 64 processes, adding OpenMP threads is not providing any acceleration and on the contrary tends to slow-down the code. These results outline that with the same code, a given execution configuration can be strongly impacted by the executing hardware. It is of course not a surprising result, but this poses the question of what is the acceptable configuration to run a given test-case. Users are given resources, sometimes virtually unlimited when compared to their actual needs. As we have seen with our convolution test-case, when over-scaled, a benchmarks end-up being dominated by parallel overhead, meaning that sometimes adding several cores only contributes to a negligible performance increase if not to a decrease. In this context, MPI_Sections can provide some compact informations relatively to what happened in the code, for example, a user could realize that his code is only doing communications.

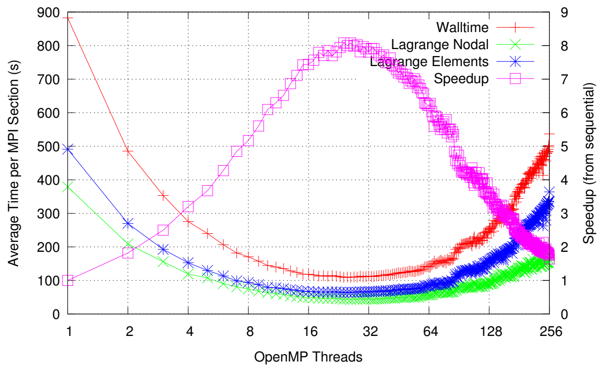


Figure 10. Lulesh Walltime and Speedup for pure OpenMP scalability on a KNL (s=48).

In order to provide a more practical illustration of how MPI_Sections can be used to pinpoint performance bottlenecks, we propose to consider Figure 10 which is directly extracted from Figure 9 (single MPI process case). We see that on the KNL parallelism budget is rapidly exhausted. Lagrangian code sections first decrease in time up to a point (at 24 threads) where their duration starts to increase. At this very point, that we denote as the *inflexion point*, the parallel overhead associated with the addition of a new thread starts to dominate. This *inflexion point* defines a speedup boundary just as a sequential fraction in Amdahl's law. We can easily demonstrate it if we apply the partial speedup bounding equation considering the sequential time T_s and the two Lagrange phases T_{p_i}, we have : $S(n = 24, s = 48) \leq \frac{T_s}{T_{p_i}} = \frac{882.48}{43.84+64.29} = 8.16\times$. As the Lagrangian sections accounts for most of the execution time, this value is very close to the actual Speedup at 24 threads which is of $8.08\times$. In fact, each section is individually bounding the speedup, and for example, `LagrangeElements` itself on the inflexion point imposes a speedup boundary of $\frac{882.48}{64.29} = 13.72\times$.

More generally, any section which duration stops decreasing with the number of threads immediately defines an upper bound on the speedup. This dynamic aspect considering sections which have exhausted their parallelism budget such as in Figure 10 is the main justification for the systematic phase outlining proposed in this paper through an MPI abstraction. Ideally, an execution configuration where the main computing section is beyond its inflexion point should never be ran as it does not productively use computing resources. A such configuration should either be limited to a given number of cores or required to augment its problem size or complexity to delay the inflexion point. This directly poses the question of scaling itself, we have shown that a given parallel algorithms (when featuring parallel overhead) is bound to have a finite speedup, and therefore an inflexion point. Consequently, if memory per core is decreasing, or just finite, the only way to scale infinitely is to use computationally expensive algorithms as the sole increase of the problem size is constrained by the memory per core budget.

5.3. On the Use of MPI_Sections

We shown that such a simple section-based analysis can provide an immediate understanding of scalability bounds relatively to macroscopic code regions. We believe that all HPC codes would gain in exposing their regions of interest through a standardized interface, allowing support tools to enrich instrumented data with Section context. A debugger would tell you that the *bug* is in the "communication" section of "load-balancing", for example. A profiler would propose a profile breakdown over sections and as introduced potential balancing information. A temporal trace viewer such as Vampir [11] would merge fine-grained trace-events per sections to provide a coarse-grain overview of section instances before zooming in, enhancing GUI scalability.

Moreover, the standard nature of MPI would allow such calls to be inserted more willingly by end-users as it would not require additional libraries unlike some approaches already provided by performance tools. This would outline computing phases in a unified manner, directly opening interesting analysis and optimizations for performance tools as we shown through these two simple examples. Besides, doing so in a runtime with user's support is much simpler than automatic approaches developed by performance tools to extract this kind of informations from raw data. Eventually, providing an alternative representation linked to the code can be profitable for some applications, for example C++ ones. Indeed, the code is what links programming models through the state of various threads. However, this state can be verbose in some languages featuring complex constructions (getter, setters, templates, ...). The ability to express a simpler state attached to an actual execution stack would provide tool with new data-management opportunities, further enhancing instrumentation's scalability.

6. Related Work

Instrumenting hybrid applications is a common problem addressed by most performance tools for common programming models such as OpenMP and CUDA.

For example, Vampir Trace [11], Scalasca [12], TAU [13] and Periscope [14] can instrument hybrid MPI+OpenMP application through the joint profiling and trace collection interface provided by Score-P [15]. Instrumentation is done through the Opari [16] source-to-source instrumenter. Code transformation which can pose issues, for example, on large C++ codes which are too complex for automatic instrumentation. Other approaches were developed at multiple levels [17] including binary ones [18], [19], [20] in order to extract OpenMP constructs which are injected at compilation time. These approaches also encountered natural limitations due to the inherent complexity associated with binary analysis. Moreover, it is impossible to be exhaustive, for example considering a static parallel loop which can be fully inlined by the compiler. Acknowledging the need for a unified instrumentation layer for OpenMP, recent efforts have led to the design of a runtime-supported OpenMP Tools interface (OMPT) [21]. This callback and sampling oriented interface is considered to become standard in OpenMP, eventually addressing the long-lasting problem of OpenMP instrumentation. Tools such as HPCToolkit [22] and TAU can rely on this interface to profile hybrid applications. CUDA is also supported by tools such as Score-P and TAU [23] most of the time through the proprietary CUDA Profiling Tool Interface (CUPTI). It is also callback oriented and provides event and counter informations describing the work offloaded to the GPU device. Other CUDA instrumentation approaches involved library wrapping to intercept CUDA calls [24], [25].

If we compare our shared-memory runtime profiling approach based on `MPI_Section` with this related work on hybrid instrumentation, we can describe it as complementary. Indeed, as we shown, Sections can profile as a side effect the code region relying on intra-node parallelism independently from its model. The reason for this is that a time slice of any application can be seen as bounding the overall speedup as we shown in Section 2. However, it is clear that once the lack of scalability is identified tools able to specifically diagnose a given programming-model are compulsory. `MPI_Sections` and more generally phases are then a coarse-grained and model-independent manner of assessing the performance of an MPI program, embedding local parallelism in global time-slices at MPI level.

Phases outlining in HPC applications similar to `MPI_Sections` has been discussed in several contexts, for example to automatically extract only part of the execution from a performance trace, to provide coarse grained knowledge of application behavior without having to look at the code or to enhance performance tools scalability (event selectivity). Several tool-side approaches were developed to outline phases, relying on signal processing [26], control-flow analysis [27], [28] , subroutines, skeletons [29] or even at instruction level [30]. Such methods have the advantage of being transparent to the user, but this comes at the cost of their relative complexity. Instead our approach is clearly simpler to implement but involves the user. We believe that this extra burden on the programmer will remain reduced and that he is the only one able to describe *semantically* what a program section does – reason for involving him. Instrumentation framework such as Score-P [15], Scalasca [31], Likwid [32] and TAU [13] provide facilities and even API to do selective instrumentation which could be seen as Section outlining.

The IPM tool [33] provides MPI level phase outlining by relying on the `MPI_Pcontrol` function call, doing so, this tool is able to leverage the MPI interface to provide features approaching what we propose as MPI Sections. However, as the Pcontrol semantic is not defined by the

MPI standard, actions (enter and leave) have to be manually encoded and therefore dependent from the target tool. This confirms the interest in defining such phases at the MPI level, in a portable and tool-agnostic manner.

We agree that our *MPI_Section* approach does not provide more functionalities that what could be done with the individual API from each of these tools. However, our approach has the advantage of being at MPI level, possibly allowing any of the aforementioned tools to rely on it instead of using dedicated code and linking to the associated library. Moreover, collective nature of the MPI_Section connects timings to actual Speedup metrics, propriety not guaranteed for local phases when outlining a code region with a tool-specific API.

7. Conclusion

In this paper, we discussed how application scalability can be expressed in order to better expose problems that can be expected when porting applications using domain decomposition to future generation platforms with greater cores counts and reduced memory per thread. After discussing common Speedup metric boundaries and proposing a practical expression of scalability and its associated boundary denoted as *partial Speedup bounding*, we argued for the differential analysis of each code section, particularly when dealing with heterogeneous codes.

After discussing distributed-memory drawbacks in the many-core context and justifying the compulsory MPI+X shift of applications and its consequence on code performance observability, we introduced the `MPI_Section` interface as a compact way of outlining code regions in MPI applications. Such interface aims at being used by profiling tools in a unified manner to query a semantic context, in contrast to the multiple interface proposed by state of the art tools. We presented this interface as two simple function calls which asynchronous collective nature is enforced by the MPI runtime and two simple callback functions possibly overridden by a tool (at PMPI level). Then, we used a preliminary tool built on top of this interface to derive factors preventing the scalability of two representative benchmarks. One, based on image convolution, measurements shown that communications were the bounding factor despite modeled as constant per process – justifying measurements to understand complex factors such as latency jitter. In the second MPI+OpenMP benchmark Lulesh, we demonstrated the use of MPI sections to assess both OpenMP and MPI scaling at the same time, clearly pointing the most efficient point of execution and portions of the code with lesser scalability.

Despite its overall simplicity, the ability to analyze code scalability and to differentiate code regions in slices spanning on the whole parallel execution is of interest for performance tools. First, to extract an execution state with more semantic than the call-stack. Second, as model-mixing is leading to nested parallelisms which can be complex to measure and model. In particular, understanding the scalability of an hybrid section can be a challenging task. The `MPI_Section` solution is then an attempt to devise a standard phase outlining process relying on the widespread MPI interface. Moreover, we have shown that due to the embedding of shared-memory models inside the MPI context, profiling MPI sections was sufficient to derive hybrid speedup metrics, effectively characterizing an MPI+X program behavior solely through MPI calls.

8. Future Work

This work and the associated profiling interface are to be released in open-source in the MALP profiling tool [34]. We are in the process of developing an MPI_Section analysis interface describing the load-balancing of Sections as shown in Figure 3.

In this paper we have shown that a given parallel program section generally presents an inflexion point bounding its speedup. If we now consider a large application with multiple sections featuring various inter-dependent algorithms, we would like to explore the possibility of dynamically restraining parallelism for non-scalable sections – investigating potential improvements for the overall computation.

References

[1] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings Redwood City, CA, 1994, vol. 400.

[2] A. B. Downey, *A model for speedup of parallel programs*. University of California, Berkeley, Computer Science Division, 1997.

[3] X.-H. Sun and Y. Chen, "Reevaluating Amdahls law in the multicore era," *Journal of Parallel and Distributed Computing*, vol. 70, no. 2, pp. 183–188, 2010.

[4] X.-H. Sun and L. M. Ni, "Another view on parallel speedup," in *Supercomputing'90., Proceedings of*. IEEE, 1990, pp. 324–333.

[5] X.-H. Sun and J. L. Gustafson, "Toward a better parallel performance metric," *Parallel Computing*, vol. 17, no. 10-11, pp. 1093–1109, 1991.

[6] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[7] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[8] A. H. Karp and H. P. Flatt, "Measuring parallel processor performance," *Communications of the ACM*, vol. 33, no. 5, pp. 539–543, 1990.

[9] J.-B. Besnard, A. Malony, S. Shende, M. Pérache, P. Carribault, and J. Jaeger, "An MPI halo-cell implementation for zero-copy abstraction," in *Proceedings of the 22Nd European MPI Users' Group Meeting*, ser. EuroMPI '15. New York, NY, USA: ACM, 2015, pp. 3:1–3:9. [Online]. Available: http://doi.acm.org/10.1145/2802658.2802669

[10] D. Holmes, K. Mohror, R. E. Grant, A. Skjellum, M. Schulz, W. Bland, and J. M. Squyres, "MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016. New York, NY, USA: ACM, 2016, pp. 121–129. [Online]. Available: http://doi.acm.org/10.1145/2966884.2966915

[11] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis tool-set," in *Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 139–155. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68564-7_9

[12] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 702–719, Apr. 2010. [Online]. Available: http://dx.doi.org/10.1002/cpe.v22:6

[13] S. Shende and A. D. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[14] S. Benedict, V. Petkov, and M. Gerndt, *PERISCOPE: An Online-Based Distributed Performance Analysis Tool*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–16. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11261-4_1

[15] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony *et al.*, "Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, TAU, and vampir," in *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

[16] B. Mohr, A. D. Malony, S. Shende, and F. Wolf, "Design and prototype of a performance tool interface for openmp," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 105–128, 2002. [Online]. Available: http://dx.doi.org/10.1023/A:1015741304337

[17] D. Lorenz, B. Mohr, C. Rössel, D. Schmidl, and F. Wolf, *How to Reconcile Event-Based Performance Analysis with Tasking in OpenMP*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 109–121. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13217-9_9

[18] L. DeRose, B. Mohr, and S. Seelam, *Profiling and Tracing OpenMP Applications with POMP Based Monitoring Libraries*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 39–46. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-27866-5_6

[19] A. S. Charif-Rubial, D. Barthou, C. Valensi, S. Shende, A. Malony, and W. Jalby, "MIL: A language to build program analysis tools through static binary instrumentation," in *20th Annual International Conference on High Performance Computing*, Dec 2013, pp. 206–215.

[20] J. Jaeger, P. Philippen, E. Petit, A. Charif Rubial, C. Rössel, W. Jalby, and B. Mohr, "Binary instrumentation for scalable performance measurement of OpenMP applications," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, ser. Advances in Parallel Computing, M. Bader, A. Bode, H.-J. Bungartz, M. Gerndt, G. R. Joubert, and F. J. Peters, Eds., vol. 25. IOS Press, Mar. 2014, pp. 783–792. [Online]. Available: http://ebooks.iospress.nl/publication/35953

[21] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, *OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 171–185. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40698-0_13

[22] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 685–701, Apr. 2010. [Online]. Available: http://dx.doi.org/10.1002/cpe.v22:6

[23] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb, "Parallel performance measurement of heterogeneous parallel systems with GPUs," in *2011 International Conference on Parallel Processing*, Sept 2011, pp. 176–185.

[24] H. Brunst, D. Hackenberg, G. Juckeland, and H. Rohling, *Comprehensive Performance Tracking with Vampir 7*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 17–29. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11261-4_2

[25] R. Dietrich, T. Ilsche, and G. Juckeland, "Non-intrusive performance analysis of parallel hardware accelerated applications on hybrid architectures," in *2010 39th International Conference on Parallel Processing Workshops*, Sept 2010, pp. 135–143.

[26] M. Casas, R. M. Badia, and J. Labarta, "Automatic phase detection of MPI applications," in *PARCO*, vol. 15, 2007, pp. 129–136.

[27] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *Micro, IEEE*, vol. 23, no. 6, pp. 84–93, 2003.

[28] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGOPS Operating Systems Review*, vol. 36, no. 5, pp. 45–57, 2002.

[29] R. Susukita, H. Ando, M. Aoyagi, H. Honda, Y. Inadomi, K. Inoue, S. Ishizuki, Y. Kimura, H. Komatsu, M. Kurokawa *et al.*, "Performance prediction of large-scale parallell system and application using macro-level simulation," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 20.

[30] J. E. Smith and A. S. Dhodapkar, "Dynamic microarchitecture adaptation via co-designed virtual machines," in *Solid-State Circuits Conference, 2002. Digest of Technical Papers. ISSCC. 2002 IEEE International*, vol. 1. IEEE, 2002, pp. 198–199.

[31] F. Wolf, B. J. Wylie, E. Abrahám, D. Becker, W. Frings, K. Fürlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore *et al.*, "Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications," in *Tools for High Performance Computing*. Springer, 2008, pp. 157–167.

[32] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE, 2010, pp. 207–216.

[33] K. Fuerlinger, N. J. Wright, and D. Skinner, "Effective performance measurement at petascale using IPM," in *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, Dec 2010, pp. 373–380.

[34] J. B. Besnard, M. Pérache, and W. Jalby, "Event streaming for online performance measurements reduction," in *2013 42nd International Conference on Parallel Processing*, Oct 2013, pp. 985–994.