Resource-Management Study in HPC Runtime-Stacking Context

Arthur Loussert, Benoît Welterlen, Patrick Carribault, Julien Jaeger, Marc Pérache, Raymond Namyst

2017 29th International Symposium on Computer Architecture and High Performance Computing

Resource-Management Study in HPC Runtime-Stacking Context

Arthur Loussert*†, Benoît Welterlen‡, Patrick Carribault*, Julien Jaeger*, Marc Pérache* and Raymond Namyst†

*CEA, DAM, DIF, F-91297 Arpajon, France

†LaBRI, Univ. Bordeaux, France

‡Bull/Atos SAS, France

***Abstract*—With the advent of multicore and manycore processors as building blocks of HPC supercomputers, many applications shift from relying solely on a distributed programming model (e.g., MPI) to mixing distributed and shared-memory models (e.g., MPI+OpenMP), to better exploit shared-memory communications and reduce the overall memory footprint. One side effect of this programming approach is runtime stacking: mixing multiple models involve various runtime libraries to be alive at the same time and to share the underlying computing resources. This paper explores different configurations where this stacking may appear and introduces algorithms to detect the misuse of compute resources when running a hybrid parallel application. We have implemented our algorithms inside a dynamic tool that monitors applications and outputs resource usage to the user. We validated this tool on applications from CORAL benchmarks. This leads to relevant information which can be used to improve runtime placement, and to an average overhead lower than 1% of total execution time.**

I. Introduction

Reaching the Exascale milestone, planned in early 2020s, will require breakthrough in both hardware and software [11]. On the hardware side, one of the major trend of the past decade was the increase of the number of cores inside processors, either regular CPUs or dedicated resource units. This leads to the rise of multicore technology with irregular accesses (NUMA nodes, cache rings or meshes...) and the advent of manycore architectures like NVIDIA GPGPUs and Intel Xeon Phi. Even if other innovations are currently on tracks for hardware development, providing an increasing number of compute units per chip is still a major evolution axis for next generations of HPC (High-Performance Computing) supercomputers.

On the software side, and more precisely regarding the parallel programming models available to exploit those compute units, MPI is widely used by most of parallel applications. However, recent studies show that scalability issues will show up at a large scale [12]. Since manycore processors exhibit shared-memory properties among a large number of cores, a natural idea to exploit these hierarchical architectures is to use *threads* to match the memory sharing capabilities of the hardware. Therefore, one typical direction is to mix MPI with a thread-based model exploiting shared-memory system leading to MPI+X programming. While mixing two models together may improve application performance, it adds a new level of complexity for the code development. Indeed, runtime libraries implementing those models are not usually designed to be interoperable with each other. Threads/processes created by different libraries are scheduled onto hardware resources by the system scheduler, most of the time without any knowledge about other existing execution flows, leading to potential cache interference, synchronization overhead or unnecessary time loss in communications. Furthermore, HPC applications like simulation codes often rely on calls to optimized libraries or different solvers to reach high performance. But each library may be parallelized with different models. For example, an MPI application could deploy a solver based on OpenMP (controlling the mix of two runtime libraries: MPI and OpenMP) and then call a second solver parallelized with Intel TBB within the same timestep. This would lead to deal with 3 models at the same time. Even if each parallel programming model may be relevant for some pieces of code, mixing them creates a *runtime-stacking* context that may lead to large overhead if the configuration of each library and the global environment are not properly set.

In this context, this paper makes the following contributions: (i) identification of runtime-stacking categories to illustrate the different situations where mixing multiple parallel programming models may appear (explicitly managed by the end-user or not), (ii) exploration of runtime-stacking configurations focusing on thread/process placement on hardware resources from different runtime libraries, and (iii) design and implementation of algorithms to dynamically check the configuration of an HPC application running on a supercomputer, leading to the detection of resource usage and model mixing. We implement these algorithms in a software tool and show how placement may impact overall performance of a parallel application. This tool introduces almost no time overhead for the target code and we validate it on some CORAL benchmarks [1].

This paper is organized as follows: Section II presents related work sorted in runtime-stacking categories while Section III details the different configurations of runtime stacking from the compute-unit point-of-view. After explaining how runtime stacking context may appear and what hap-

978-1-5090-1233-6/17 $31.00 © 2017 IEEE
DOI 10.1109/SBAC-PAD.2017.30

pens at execution time on hardware, Section IV describes our proposed algorithms to detect the current runtime-stacking configuration and check for resource usage. Finally, we present some experimental results in Section V, through the implementation of our algorithms in a tool before concluding in Section VI.

II. Related Work

As mentionned previously, relying exclusively on MPI programming model may lead to scalability issues [12], which explains why MPI is increasingly mixed with thread-based models to improve overall performance. We call the composition of runtime libraries *runtime stacking*. But this stacking is only one example where mixing appears. Indeed, calling external libraries or relying on other models (more abstracted) may involve runtime stacking situations. To explore those situations, we introduce *categories* as depicted in Figure 1.

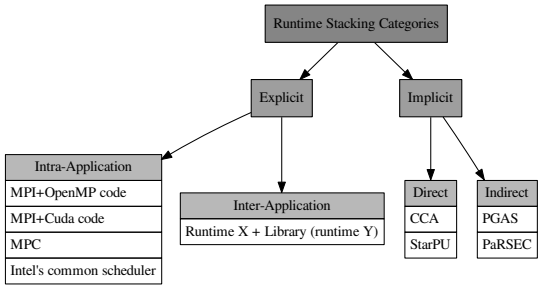

Figure 1. Runtime Stacking Categories

The most common way to lead to runtime-stacking context in a code is through an *explicit* approach. It means that application developer might be aware of this stacking situation and can have tuning knobs to change the configuration and improve performance. Inside this *explicit* branch, *intra-application* stacking groups the applications that explicitly call different runtime libraries, for example in an hybrid MPI+OpenMP code. Thus stacking is directly exposed by the programmer through MPI function calls and OpenMP directives. Yet, even if relying on various optimized parallel runtime libraries would seem natural, most applications only use one or two of them at a time. Indeed, models were not designed to run concurrently on the same resources. This problem is known as the composability problem [8]. There exist initiatives that help users writing optimized hybrid code. For example, MPC [5], [9] is a framework that provides a unified parallel runtime designed to improve the scalability and performance of applications running on HPC clusters. It allows mixed-mode programming models and efficient interaction with the software stack by providing its own MPI, OpenMP and pthread implementations, based on an optimized user-level scheduler. However, it does not give the user any feedback about resource usage. With the same objective of composing models, Intel has been using a common runtime system basis [7], [10], to limit the interference between their runtimes. If used in the same application, Intel TBB and Intel Cilk can run concurrently by sharing the underlying task scheduler, and thus avoid thread oversubscription.

Another way to exhibit stacking is through calls to external libraries. It corresponds to the *explicit inter-application stacking* leaf on Figure 1. For example, writing an MPI code and calling libraries based on OpenMP (e.g., BLAS MKL) or Intel TBB is fairly common. In this case stacking is indirect, as calls to different parallel models are made in external libraries. However, by knowing the library implementation (or at least the parallel programming model they rely on), stacking can still be explicitly observed. Note that such libraries often try to improve performance by bypassing the system scheduler through direct allocation and binding of execution flows. As each library may be unaware of other resource utilization, the target code will potentially exhibit poor performance due to those libraries interferences.

On the other hand, the *implicit* branch of the tree encloses situation where runtime stacking implicitly appears. This is the case when programmers rely on unified or abstracted platforms designed to help them compose pieces of codes. The Common Component Architecture (CCA) [2] is an example of implicit stacking. Components are black boxes used to build an application. Each component can be parallelized directly with regular models (e.g., MPI, OpenMP or NIVIDA CUDA) but when using them, we may or may not know how they were optimized and which runtimes were used. We call these techniques *implicit direct stacking* in Figure 1. StarPU is an other example of platform that helps users stack models [3]. The runtime-system goal is to manage parallel tasks over heterogeneous hardware. StarPU relies on an hypervisor to dynamically chose which implementation of a kernel will be more suitable for the target hardware resources. Moreover, it uses a dynamic resource allocation with *scheduling contexts* [6]. The use of a global hypervisor may mitigate scheduling problems, however tasks are still coded with classic runtimes, thus creating situations where runtime stacking issues may appear.

Finally, the last category called *implicit indirect stacking* represents approaches where end-users may have difficulties to know which parallel programming models the application will eventually use. This category includes PGAS languages that abstract the way to communicate and share memory, relying on different models to improve performance (for example, a PGAS library implementation may rely on MPI for inter-node communication and regular threads for intra-node synchronizations). In these situations, the implementation is in charge of selecting the best combination and

helps the application end-user to configure the resource usage on supercomputers. This category also includes new approaches like PaRSEC [4], an event-driven runtime that handles task scheduling and data exchanges which are not explicitly coded by the developers.

The categories highlight the fact that there are many situations that may exhibit runtime stacking. Each of the approaches described in this section is relevant and has its own advantages/drawbacks depending on the target hardware resources and the current state of the parallel application. The common aspect is that eventually, multiple model implementations may coexist during the execution of the application leading to poor resource usage if parameters are not correctly set. Therefore, there is a need for tools which would help the end-user to check the dynamic behavior of the target application and extract the resource usage. To do so, we propose to derive relevant stacking configurations from the resource usage point of view.

III. Runtime Stacking Configurations

Focusing on the *explicit* branch of categories from Figure 1, this section presents our first contribution: the study of different runtime-stacking configurations that may appear when a hybrid application (with multiple programming models) runs on a cluster. Indeed, at execution time, the different threads/processes are scheduled on computational resources and the runtime libraries (corresponding to programming models) are scattered and executed across the machine. Depending on many parameters (including the machine configuration, scheduler policy, runtime implementation, resource reservation method, hints, environment variables...), the eventual placement of threads from the whole application may vary. Based on this placement and resource usage, we define the notion of *runtime-stacking configurations* to represent how execution flows (threads and processes) are scheduled on a target machine. The main goal of these configurations is to help the user to understand how the underlying runtime libraries interact with each other in the application.

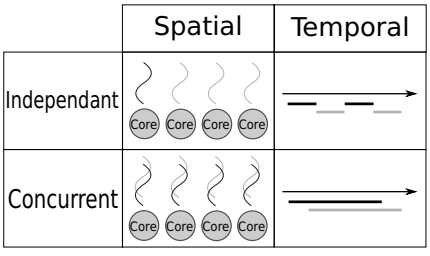

Figure 2. Runtime-Stacking Configuration

Figure 2 presents the different configurations based on the resource concurrency (rows in the figure) according to space and time (columns in the figure). Thus, two runtime libraries can be *spatially independent* if they deploy execution flows on disjoint resources. The figure illustrates this configuration with a simple example: 4 cores are depicted with 2 parallel programming models used by an application (the black model and the grey one). In the independent/spatial cell, these models create a total of 4 execution threads: three of them belong to the grey model while the last one is managed by the black runtime. Because each flow is scheduled on separate resources and each compute unit is busy with one execution flow, we say that this situation is *spatially independent*. This is mainly the first target configuration when developing and optimizing an hybrid application exhibiting runtime stacking. In this case, there is basically no interaction between the two available runtime libraries at the hardware level because the set of resources is disjoint.

On the other hand, if some execution flow competes for the same resource, we enter the configuration called *spatially concurrent*. Figure 2 depicts an example of such configuration by oversubscribing every core with two threads: one coming from the black programming model and one managed by the grey one. This configuration may involve compute-cycle stealing and, therefore, lead to performance loss during execution. One key parameter can be the *wait policy* which drives the way each programming model will put the corresponding thread asleep when not active. For example, in OpenMP, it is possible to choose between *active* and *passive* mode. With active mode, each thread waiting for work will consume CPU cycles. It would result in better reactivity when starting a new parallel region, but it may reduce the performance of the other model performing computation on the same cores at the same time. On the other hand, passive mode allows better interoperability but may lead to poor performance because threads may take more time to wake up for the next parallel region. There are different situations where this configuration may appear. First of all, users may ask as many threads as the number of available cores on the target compute nodes (with environment variables like `OMP_NUM_THREADS` or some parameters of job manager). Another scenario is a bad resource allocation for the different runtime libraries. Indeed, if all programming models are not aware from each other, they may want to deploy their execution flow on the whole node.

In addition to the spatial configurations, the temporal scheduling is also of interest. Two models are *temporally independent* if there is no overlap in their creation-destruction time frame. This is the case when a code performs multiple repetitive calls to an optimized parallel library. Each call will spawn and deploy library-related threads, but each instance of these runtimes will be independent from each other. This example is depicted in Figure 2 where the arrow represents the evolution of time and two bars (black and grey) show how the different programming models overlap. The notion of *temporally independent* flows leads to non-overlapping

bars. Conversely two runtimes are said to be *temporally concurrent* if they can be scheduled in the same time frame. This is the case when a MPI+OpenMP hybrid code makes MPI calls from within OpenMP parallel sections for example. Once again, Figure 2 illustrates this configuration with two bars (black and grey) overlapping over time.

During the entire execution of an application, multiple *runtime stacking configurations* may alternate. These configurations may also vary from one execution to another. By looking at how execution flows were scheduled, we can determine if there is a contention issue with parallel models, or if every instance was scheduled on its own set of resources without interference. Knowing which configuration appears is also a valuable information when developing and optimizing a code, as spatially independent runtimes is often what programmers are looking for. We propose in the next section to detect the execution configuration through resource-based algorithms.

IV. Algorithms Detecting Resource Usage

This section presents the second contribution of this paper: algorithms to detect the configuration of a target parallel application. To obtain the big picture of what happened at execution time, and help optimizing flow placement as well as resource usage, we present algorithms which take as input execution information of an application (through traces or online events) and determine the corresponding resource usage. They output warnings if the load of the machine is detected as non-optimal (overloaded resources or idle resources).

We separate this section into two main algorithms. First of all, we detect wrong usage of resources through execution flows and then we focus on each resource to check if each of them is busy or not. In the examples as well as experimental results, we focused on threads (flows) and cores (resources) but note that these algorithms could be used to detect misuses of other hardware resources (compute units, memory, IO components) from different flows (instructions, data, messages).

Data: $Flow = F_0, ..., F_n$, $Resource = R_0, ..., R_n$
$S \leftarrow \emptyset$;
foreach $F \in Flow$ **do**
  $S \leftarrow S \cup R_F$;
end
if $(\sum_{i=0}^{n} |F_i|) \neq |S|$ **then**
  produce warning;
end

Algorithm 1: Flow-Centric Algorithm

Algorithm 1 focuses on execution *flows* (i.e., groups of instructions executing on target resources). *Flows* require resources to progress. However all resources may not be accessible (depending on the parameters set by the user and the global system environment). For this purpose, we define R_F as the entire set of hardware resources that the *flow* F can access during execution. Taking as input these flows and the corresponding available resources, the algorithm iterates on each *flow* and creates a set containing all *resources* accessible by all *flows*. Then, if the cardinality of this resulting set S (i.e., number of available resources for the application) is different from the sum of all *flows* cardinality, the resource **reservation** may be suboptimal.
For example we may execute a 2-process code with four cores per process on a eight-core node. If both processes spawn more than four threads, resources will be overloaded. In this case Algorithm 1 would produce a warning informing the user that processes created too many threads according to available resources. Similarly if processes created less than four threads, Algorithm 1 would produce a warning about idle resources.

Data: $Flow = F_0, ..., F_n$, $Resource = R_0, ..., R_n$
foreach $R \in Resource$ **do**
  $S \leftarrow \emptyset$;
  foreach $F \in Flow$ **do**
    if $R \in R_F$ **then**
      $S \leftarrow S \cup F$;
    end
  end
  if $|S| \neq 1$ **then**
    produce warning;
  end
end

Algorithm 2: Resource-Centric Algorithm

But the previous algorithm alone is not enough to detect all situations where different resource usage may appear. Thus, Algorithm 2 focuses on resources. It traverses through each resource R and checks, for each flow F, if R is included in the set of resources R_F that F can access. If more than one *flow* can access one specific *resource*, or if a *resource* can't be accessed, then the main **repartition** may be suboptimal.

For example if we launch a 2-process code with four cores per process on a 8-core node and disabled process binding, the job manager may allocate the same cores for both processes. It would lead to an execution scheduling eight threads on four cores, keeping four cores idle. In this situation, Algorithm 2 detects that four cores are used by two different processes and produces 2 warnings: one about overloaded resources and another one regarding idle cores.

We might note that depending on what *flows* are given when running algorithm 2, results may vary. For example by launching a multi-threaded process, each thread created

might be able to access all resources allocated to the process. Thus by inputting threads as *flows*, algorithm 2 would detect that each resource is accessible by multiple threads and would generete a warning. On the other hand if the multi-threaded process is inputted as a *flow* the algorithm would look at the number of threads created and if it is equal to the number of cores available, there are no warning to be printed. Thus in practice this algorithm might be used on different granularities when applicable and results combined. Moreover if information is not available at thread level, only applying the algorithm to the whole process will still provide valuable information.

These algorithms used together detect misuses coming both from resource allocation and flow repartition giving an accurate identification of the source of potential resource misuse.

According to configurations introduced in Section II, Algorithms 1 and 2 mainly focus on spatial stacking. Indeed, when receiving a warning the user will have information on both the resource usage (idle or overloaded resources) and the spatial configuration (concurrent accesses). Yet these algorithms do not take temporality into account. This would require to grab more specific information about programming models (for example waiting policy). This extension is left for future work.

V. Implementation and Experimental Results

The algorithms presented in Section IV take subsets of resource allocation as input. To produce and process these input data, we designed a tool made of 2 parts: (i) a dynamic part which oversees the execution of an application and produces logs containing information about threads and runtime libraries, and (ii) a post-mortem part which applies the resource-usage algorithms. We tested our tool on CORAL benchmarks [1]: AMG2013, LULESH, miniFE and Nekbone. The testbed configuration is composed of a cluster with 2-socket 32-core hyperthreaded Haswell nodes, 2-socket 16-core non-hyperthreaded Sandy Bridge nodes and Intel KNL nodes. The software environment includes OpenMPI 2.0.1 and Intel OpenMP 17.0.0.98.

A. Tool Design and Implementation

The goal of the dynamic part of the tool is to gather relevant input data that will be used by our algorithms. It collects information on threads/processes from their creation to their destruction. We implemented this part as a library which is preloaded at execution time through the `LD_PRELOAD` mechanism. Each process created during the execution of the target applicaton loads its own instance of the library, so that the tool gets access to information about all processes. The library wraps the `pthread_create` function to track all created threads and their parameters (pid, tid, name of the module calling `pthread_create`, cpuset the processes threads are allowed to run onto...). Now that the tool has access to information from all processes and threads, it can create logfiles. We create a file per process, each of them begining with all information gathered on the process at startup. Then each `pthread_create` function call adds a line in the file with information on the new thread. In order to keep track of the cores a process has access to, the tool also wraps the `set_affinity` function calls adding a line in the logfile. In addition, each line in the output trace file is written with a timestamp to provide a temporal view of the execution.

Information provided by these logs is not exhaustive, however it allows the analysis of interactions of runtime librairies with our algorithms presented in Section IV. Moreover the small amount of data collected leads to a lightweight tool that generates small traces and therefore a small overhead.

By parsing the output traces, the post-mortem part of the tool retrieves all the relevant information about processes and threads. Algorithms 1 and 2 are then applied. *Resources* are assimilated to cores and *flows* to processes and their threads. As presented in section IV, we run algorithm 2 twice, once at the process granularity and once at the thread granularity. With all information gathered from traces and algorithm result, an output is generated, giving a summary of process cpusets and thread placement (*spatial stacking configurations*) as well as information about resource usages and runtime libraries through warnings. With these information and hints, user can determine if a better resource usage is possible and how to achieve it.

B. Tool Overhead Evaluation

Table I presents the overhead of our tool when running along with target CORAL benchmarks on different hardware configurations. The first two columns show the results on Haswell nodes, and the following two present results using Intel KNL nodes. The first column presents the results on two Haswell nodes, using sixty-four MPI tasks (thirty-two per node), and two OpenMP threads per task (two per hyperthreaded core). The second presents the results of runs using thirty-two Haswell nodes, using sixty-four MPI tasks (two per node, one per socket), and thirty-two OpenMP threads per task (using all hyperthreads of a socket per task). The third column shows results using one KNL node, sixty-four MPI tasks and four OpenMP threads per task. The last column presents results using eight KNL nodes, sixty-four MPI tasks and thirty-two OpenMP threads per task. This table shows an overhead lower thant 0.1% using our tool. Furthermore, this overhead seems stable accross applications running on the same configurations. This is due to the fact that our tool only instruments a couple of functions. As most of the time in HPC applications, cpusets of processes are set once and for all at their creation and threads are used by multiple librairies without them being destroyed and re-created. Thus the interference of our tool can almost only be seen at startup.

		2 Haswell nodes 64 MPI tasks 2 OpenMP threads per task	32 Haswell nodes 64 MPI tasks 32 OpenMP threads per task	1 KNL nodes 64 MPI tasks 4 OpenMP threads per task	8 KNL nodes 64 MPI tasks 32 OpenMP threads per task
Lulesh	with tool (s)	46.790	2668.745	398.135	213.695
	without tool (s)	46.945	2672.100	398.580	213.760
	overhead (%)	<0.01	<0.01	<0.01	<0.01
miniFE	with tool (s)	17.789	303.325	66.728	102.272
	without tool (s)	18.805	303.733	68.350	102.331
	overhead (%)	0.06	<0.01	0.02	<0.01
AMG	with tool (s)	33.476		64.939	
	without tool (s)	34.532		65.553	
	overhead (%)	0.03		0.01	
Nekbone	with tool (s)	28.255	12.623	98.159	40.920
	without tool (s)	29.219	13.217	98.629	41.174
	overhead (%)	0.03	0.05	<0.01	<0.01

Table I
EVALUATION OF TOOL OVERHEAD ON VARIOUS ARCHITECTURES

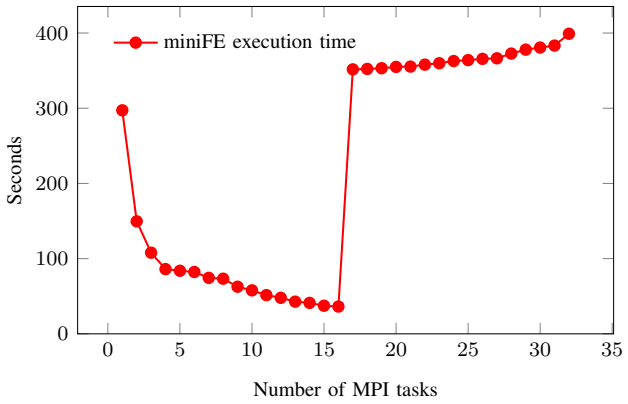

Figure 3. Execution time of miniFE with different number of MPI tasks

C. Benchmark Evaluation

In order to show how resource allocation and resource placement have an impact on the performances of applications, we ran four CORAL benchmarks with different allocation and placement methods on our testbed. Figure 3 presents the total execution time of the miniFE application on a Sandybridge node (16 cores without hyperthreading) when the number of MPI tasks grows and the problem size remains the same. We can see that from 1 to 15 cores while the node is underloaded, the execution time of the simulation decreases. It reaches its best time when the node is full, at 16 MPI tasks, and then execution time starts to increase when we overload the node with more tasks than available core (from 17 to 32 tasks). This conclusion that better performances are obtained when using the right number of cores is obvious. However, finding the best configuration is not always easy, and it is often hard to indentify the source of a lack of performances when the error comes from allocation and placement of tasks and threads. Indeed neither error nor warning are produced by the compiler or at execution.

This experiment shows that the resource allocation has an impact on the performances of a simulation. Our algorithms presented in Section IV can help finding a resource allocation configuration using the maximum of resources. For example, using our tool with each configuration from Figure 3 produces warnings except when using 16 tasks, which is the configuration using the architecture at its fullest without overloading the node. With an underloaded node, e.g. with only one MPI task, the tool produces the following warnings:

```
+ Node 0: 16 cores available, 1 process created.
          Process 0 created 1 thread on cpu(s) [0]
  ## WARNING ## Cores [1 - 15] may stay idle during
       execution (underloaded resources)
Analysis conclusion: 1 warning(s)
```

In the same manner, an execution overloading nodes with 32 MPI tasks outputs the following warnings:

```
+ Node 0: 16 cores available, 32 process created.
          Process 0 created 1 thread on cpu(s) [0]
          Process 1 created 1 thread on cpu(s) [1]
          [...]
          Process 16 created 1 thread on cpu(s) [0]
          Process 17 created 1 thread on cpu(s) [1]
          [...]
  ## WARNING ## Cores [0 - 15] may be used by more than
       one process/thread (overloaded resources) (spatial-
       concurrent configuration)
Analysis conclusion: 1 warning(s)
```

Figures 4 to 6 presents the impact of tasks and threads placement on application performances. For these experiments we used three CORAL benchmarks: Lulesh, minFE and AMG2013, and three target architectures: Haswell nodes with hyperthreaded cores, Sandybridge nodes without hyperthreading, and one KNL node with four hyperthreads per core. For each application, we varied the runtimes options. The 'scatter', 'compact,0' and 'compact,1' correspond to Intel OpenMP 'KMP_AFFINITY' options. The 'scatter' option means that all OpenMP threads are spaced as much

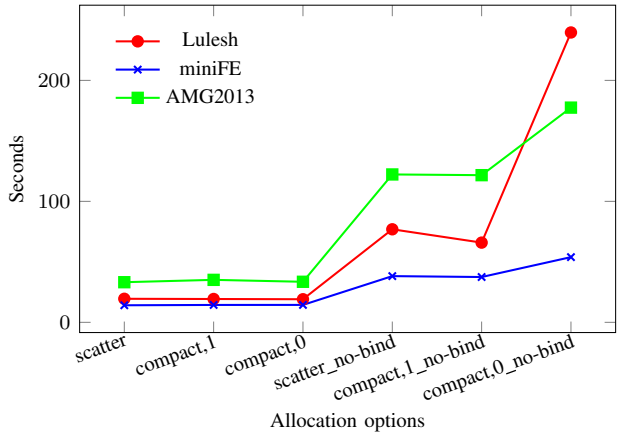

Figure 4. Execution time of CORAL Benchmarks on Haswell nodes

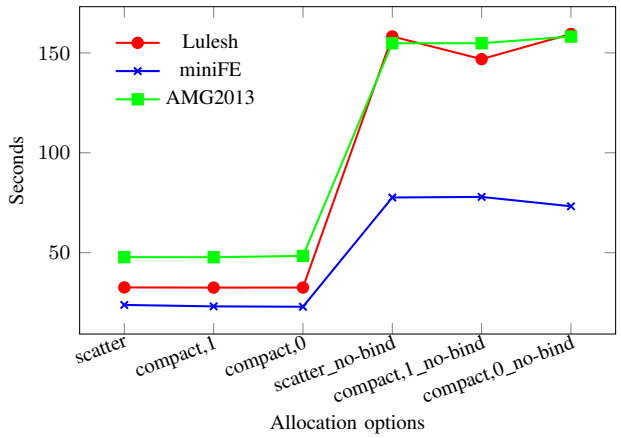

Figure 5. Execution time of CORAL Benchmarks on Sandybridge nodes

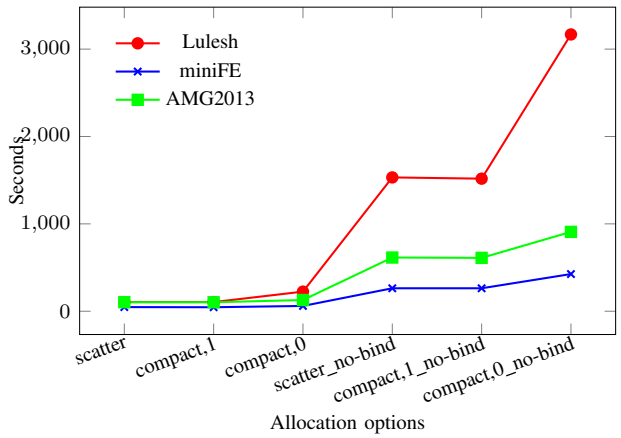

Figure 6. Execution time of CORAL Benchmarks on a KNL node

as possible on cores (to maximize cache size and memory bandwidth), the 'compact,0' places one thread per logical core (including hyperthreads), and the 'compact,1' option fills physical core first. The 'no-bind' option indicates that process binding was disabled in the job manager. In our case, it means that all processes are given the same cpuset, resulting in overloaded ressources and idle ones. Figure 4 shows the results of each benchmarks using two Haswell nodes, eight MPI tasks (two per node) and sixteen OpenMP threads per process effectively using all cores and all hyperthreads of both nodes. Figure 5 exposes the results of the same benchmarks on two Sandybridge nodes, eight MPI tasks (two per node) and eight OpenMP threads per process effectively using all cores of both nodes as hyperthreads are not activated. Finally, Figure 6 presents the results of the benchmarks on one Intel KNL node with eight MPI tasks and eight OpenMP threads per task using all cores but half of the hyperthreads of the node.

These graphs exhibit two behaviors. First when the process binding is disabled, execution time increases on all architectures. This was expected as we use less cores for the same computations and the same number of tasks/threads. By disabling the binding the number of threads per core highly increases seriously impeding performances. This is especially strong on KNL nodes which use a lot of hyperthreads. Note that in these cases, our tool produces the warnings to the user about a clumsy use of resources:

```
+ Node 0: 32 cores available, 4 process created.
          Process 1 created 8 thread on cpu(s) [0-8]
          Process 2 created 8 thread on cpu(s) [0-8]
          Process 3 created 8 thread on cpu(s) [0-8]
          Process 4 created 8 thread on cpu(s) [0-8]
  ## WARNING ## Cores [0 - 8] may be used by more than one
       process/thread (overloaded resources) (spatial-
       concurrent configuration)
  ## WARNING ## Cores [9 - 31] may stay idle during
       execution (underloaded resources)
Analysis conclusion: 2 warning(s)
```

Furthermore we can see that the OpenMP environment may have an impact on performances. For example, on Haswell nodes (figure 4), using the 'compact,1' option with the AMG2013 benchmarks adds a 6% overhead to the execution time compared to the same execution with the 'compact,0' option. On the other hand, running the Lulesh benchmark using the 'compact,0' option on KNL nodes (figure 6) more than doubles the execution time compared to the same execution with the 'scatter' option. These results are explained by the fact that these options change the placement and binding of threads. As all cores and hyperthreads are used with the AMG2013 benchmark on Haswell nodes, the overhead observed is probably a consequence of a different data placement intensifying the NUMA effects when using the 'compact,1' option. Note that in this case, our tool do not produce warnings as resources are busy with one execution flow. Moreover this placement will not always influe on performances in the same manner on all architectures. For example the same

benchmark on Sandybridge nodes (figure 5) exposes the best performances with the 'compact,1' option. By looking at the tool output, users could see that the cpusets of processes are spread on multiple processors but the conclusion regarding performances would come from knowledge of both architecture and application's data usage. However, when using these OpenMP options with a partially full node, the tool detects what may be wrong use of resources and produces warnings. For example using the 'compact,0' option on a KNL node more than doubles the execution time of the Lulesh benchmark (figure 6). In this case the tool would detect that only half the cores of the node are used and inform user that half the resources may stay idle during execution. Note that with the 'scatter' or 'compact,1' options the tool would also produce warnings this time informing that all cores are used but with only half the hyperthreads.

These experimental results show that allocation and placement of tasks and threads have an impact on application performances. This impact may vary depending on architectures and applications. Our tool can give user a view of resource usage and thread placement, as well as it produces warning when a potential misuse of resources is detected. This kind of analysis is important as allocation and placement are not checked by application or system, even when obvious misuses occur, which may lead to bad performances.

VI. Conclusion & Future Work

This paper explored a solution to the increasing number of compute units per chip as well as the rise of heterogeneous architectures. This solution, runtime stacking, provides a frame to optimize codes, increase the scalability of existing applications and extract the most out of complex architectures. However stacking also requires more complex resource management. We define state-of-the-art categories of runtime stacking including explicit hybridization of programming models as well as indirect use of multiple runtime systems. We study runtime stacking categories, design and implement algorithms to observe and improve resource management when mixing multiple parallel programming models like MPI and OpenMP. We validate our assertion of the importance of resource management by experimenting on real applications from the CORAL benchmarks.

For future work, we plan to improve our analysis tool to provide optimal resource reservation arguments depending on runtime stacking categories and desired configurations (with potential hints from user). We also plan to design a centralized tool that would dynamically manage resources by looking at runtime stacking configurations through our algorithms. Multiple paths can be explored: global hypervisor, directives from user that could define a range of needed resource per code section, or prediction. All of these have prerequisites: runtime libraries need to be able to share resources, either by exchanging topologies and negotiating resources, or by involving the scheduler which will need to be able to manage runtimes to allow them more or less resources dynamically.

References

[1] Coral benchmarks codes. https://asc.llnl.gov/CORAL-benchmarks/.

[2] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of HPDC 1999*, pages 115–124. IEEE, 1999.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] Rajive Bagrodia, Richard Meyer, Mineo Takai, Yu-an Chen, Xiang Zeng, Jay Martin, and Ha Yoon Song. Parsec: A parallel simulation environment for complex systems. *Computer*, 31(10):77–85, 1998.

[5] Patrick Carribault, Marc Pérache, and Hervé Jourdren. Enabling low-overhead hybrid mpi/openmp parallelism with mpc. In *Proceedings of IWOMP 2010*, volume 6132 of *Lecture Notes in Computer Science*, pages 1–14. Springer Berlin Heidelberg, 2010.

[6] Andra Hugo, Abdou Guermouche, Pierre-André Wacrenier, and Raymond Namyst. Composing multiple starpu applications over heterogeneous machines: a supervised approach. *The International Journal of High Performance Computing Applications*, 28(3):285–300, 2014.

[7] Andrey Marochko. Tbb 3.0 task scheduler improves composability of tbb based solutions. part 1. https://software.intel.com/en-us/blogs/2010/05/13/tbb-30-task-scheduler-improves-composability-of-tbb-based-solutions-part-1. Accessed: 2017-06-12.

[8] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with lithe. *ACM Sigplan Notices*, 45(6):376–387, 2010.

[9] Marc Pérache, Hervé Jourdren, and Raymond Namyst. MPC: A unified parallel runtime for clusters of NUMA machines. In *Proceedings of Euro-Par 08*, pages 78–88, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] Mark Sabahi. Getting code ready for parallel execution with intel parallel composer. https://software.intel.com/en-us/articles/getting-code-ready-for-parallel-execution-with-intel-parallel-composer. Accessed: 2017-06-12.

[11] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *International Conference on High Performance Computing for Computational Science*, pages 1–25. Springer, 2010.

[12] Rajeev Thakur, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing Lusk, and Jesper Larsson Träff. MPI at exascale. *Proceedings of SciDAC*, 2:14–35, 2010.