An On-line Performance Introspection Framework for Task-based Runtime Systems*

Xavier Aguilar[1], Herbert Jordan[2], Thomas Heller[3], Alexander Hirsch[2], Thomas Fahringer[2], and Erwin Laure[1]

[1] KTH Royal Institute of Technology, Department of Computational Science and Technology and Swedish e-Science Research Center (SeRC), Lindstedvägen 5, 10044 Stockholm, Sweden
[2] University of Innsbruck, Department of Computer Science, Technikerstrasse 21a, 6020 Innsbruck, Austria
[3] Friedrich-Alexander-University Erlangen-Nuremberg, Department of Computer Science, Martenstr. 3 91058 Erlangen

Abstract. The expected high levels of parallelism together with the heterogeneity and complexity of new computing systems pose many challenges to current software. New programming approaches and runtime systems that can simplify the development of parallel applications are needed. Task-based runtime systems have emerged as a good solution to cope with high levels of parallelism, while providing software portability, and easing program development. However, these runtime systems require real-time information on the state of the system to properly orchestrate program execution and optimise resource utilisation. In this paper, we present a lightweight monitoring infrastructure developed within the AllScale Runtime System, a task-based runtime system for extreme scale. This monitoring component provides real-time introspection capabilities that help the runtime scheduler in its decision-making process and adaptation, while introducing minimum overhead. In addition, the monitoring component provides several post-mortem reports as well as real-time data visualisation that can be of great help in the task of performance debugging.

Keywords: Runtime System · Performance Monitoring · HPX · Performance introspection · Real-time Visualisation · AllScale.

1 Introduction

The increase in size and complexity of upcoming High Performance Computing (HPC) systems pose many challenges to current software. Upcoming, and already existing, HPC infrastructures contain heterogeneous hardware with multiple levels of parallelism, deep memory hierarchies, complex network topologies

* Supported by the FETHPC AllScale project under Horizon 2020.

and power constrains that impose enormous programming and optimisation efforts. Thus, new high-productivity, scalable, portable, and resilient programming approaches are needed.

Task-based runtime systems have emerged as a good solution for achieving high parallelism, performance, improved programmability, and resilience. Proof of that is the amount of new task-based runtimes that have appeared in the HPC landscape in recent years, e.g. HPX [11], Legion [3], StarPU [2], or OmpSs [5] among others.

The AllScale project tries to settle in the HPC runtime landscape as a new solution that provides a unified programming interface for parallel environments. One key aspect differentiating AllScale against other existing runtimes is that it is heavily based on recursive parallelism to diminish the amount of global synchronisation present in classical parallel approaches. Reducing thereby one of the main factors that hinders scalability.

In this paper, we give an overview of the AllScale toolchain as well as an extensive characterisation of one of its key components, the AllScale Monitoring Component. This monitoring framework provides introspection capabilities such as real-time performance data visualisation and real-time performance feedback to the AllScale Runtime Scheduler. In addition, it can also provide several performance reports suited for post-mortem performance debugging. Furthermore, even though the monitoring component presented in this paper has been especially designed for the AllScale Runtime, it can easily be decoupled and adapted to any other task-based runtime system.

The paper is structured as follows: Section 2 provides an overview of the AllScale project; the AllScale Monitoring Component is described in depth in Section 3; Section 4 provides a detailed evaluation of the monitoring component; related work is described in Section 5, and finally, conclusions and future work are presented in Section 6.

2 AllScale Overview

2.1 Vision

The AllScale project pursues the development of an innovative programming model supported by a complete toolchain: an API, a compiler, and a runtime system. AllScale is strongly based on recursive programming in order to overcome inherent limitations that nested flat parallelism presents, for example, the use of global operations and synchronisation points that limit the scalability of parallel codes. Moreover, AllScale aims to provide a unified programming model to express parallelism at a high level of abstraction using C++ templates. In that way, problems imposed by the use of multiple programming models can be mitigated, for example, the need of expertise in complementary models such as MPI, OpenMP, or CUDA. Furthermore, by using a high level of abstraction to express parallism, problems such as the need to design the algorithm based on the underlying hardware can be mitigated too.

2.2 Architecture

The AllScale toolchain is divided into three major components: an API, a compiler, and a runtime system. The AllScale API, based on the prec operator [10], provides a set of C++ parallel operations such as stencils, reductions, and parallel loop operations, as well as a set of data structures (arrays, sets, maps, etc.) to express parallelism in an unified manner. AllScale applications can be compiled with standard C++ tools, however, in order to take advantage of all the benefits of the AllScale environment, the code has to be compiled with the AllScale Compiler. The compiler, based on the Insieme source-to-source compiler [8], interprets the API and generates the necessary code to execute the application in the most optimal manner by the AllScale Runtime System. Finally, the AllScale Runtime System [9] manages the execution of the application following customisable objectives that can be defined by the user. This multi-objective optimisation combines execution time, energy consumption, and resource utilisation, and thus, improves classical approaches where self-tuning during execution is mainly based in execution time. In addition, the AllScale Runtime provides transparent dynamic load balancing and data migration across nodes, and it is the basis for the resilience, scheduler and performance monitoring components. The AllScale Runtime System builds on top of the High Performance ParalleX (HPX) runtime [11].

3 AllScale Monitor Component

The AllScale environment includes a monitoring framework with real-time introspection capabilities. Its main purpose is to support the AllScale Scheduler in the management of resources and scheduling of tasks. To this end, the AllScale Monitor Component has been designed to introduce minimum overhead while being able to continuously monitor the system and provide in real time the performance information collected. In addition, the AllScale Monitoring Component can also generate several post-mortem reports to help developers in the process of performance debugging.

3.1 Performance Data

The AllScale Monitor Component collects several performance metrics on the execution of tasks (WorkItems [8, 9] in AllScale jargon). These WorkItems, which can be hierarchically nested, are entities that describe the work that has to be performed by the runtime system. The AllScale Monitor Component takes control at the beginning and end of a WorkItem, and collects performance data such as execution time or WorkItem dependencies (what WorkItems have been spawned from within the current one). From these raw metrics, the monitoring component is also able to compute multiple derived metrics: execution time of a WorkItem and all its children; percentage of exclusive and inclusive time per WorkItem over total execution time; average execution time of all children of a WorkItem; or standard deviation of the execution time for all children of a WorkItem, among others. These derived metrics are computed on demand to minimise the amount of overhead introduced into the application when collecting the data. In other words, they are computed only if the scheduler requests them or if a performance report that includes them has to be generated.

The metrics described above are collected in an event action basis, that is, the collection is triggered by an event such as WorkItem start or stop for example. However, the AllScale Monitor Component also samples many other metrics on a per-node basis[4], for instance, WorkItem throughput, memory consumption, runtime idle rate, or CPU load, among others. The runtime idle rate is defined as the percentage of time spent by the runtime in scheduling actions against the time spent executing actual work. In addition, the AllScale Monitor Component is able to sample HPX counters (HPX internal metrics) and PAPI counters.

Power is becoming an important constraint in HPC, and therefore, the AllScale Monitoring Component can also provide power and energy consumed on x86 platforms, Power8, and Cray systems. If power and energy measurements are not available, the monitoring component is able to provide simple estimates.

3.2 Data Collection

The AllScale Monitor Component has been designed to introduce as minimum overhead as possible. When an application starts, the AllScale Runtime system creates a pool of worker threads that execute tasks as they are generated. These WorkItems, which are then monitored by the AllScale Monitor, are very small and thousands of them are executed in every run. Therefore, solutions where each thread collects and writes its data to shared data structures do not work due to heavy thread contention.

For this reason, the AllScale Monitor Component has been implemented following a multiple producer-consumer model with a double buffering scheme. All runtime worker threads measure and keep their raw performance data locally. Once their local buffers are full, they transfer the data to specialised monitoring threads that process and store such data into per-process global structures. The data have to be stored at a process level in order to facilitate the performance introspection later on. Only the specialised monitoring threads write into global data structures, and thereby, the contention is very low. Furthermore, producers and consumers utilise a double buffering scheme, that is, while producers create data in their buffers, consumers process data from an exclusive buffer that nobody else accesses. When a consumer finishes processing its data, it switches buffers with the next producer waiting to send data, and the process starts again. Even though our system has the possibility to have more than one consumer thread, we have seen in our experiments than one specialised thread per node is enough to process the amount of data generated.

The monitoring infrastructure also has another specialised thread responsible for collecting metrics sampled on a per-node basis. This thread awakes at fixed

[4] Node defined as compute or cluster node

time intervals and collects performance metrics such as WorkItem throughput, node idle rate, memory consumption, or energy consumed, among others.

The AllScale Monitor Component is also able to select the type of tasks monitored, thereby, further reducing its overhead and memory footprint. In AllScale, there are two types of WorkItems derived from the recursive paradigm exploited in the runtime: splittable and processable. Splittable WorkItems are tasks that are split and generate more tasks, they correspond to the inductive case of the recursion. Processable WorkItems correspond to the base case of the recursion, and are WorkItems that are not split anymore. The user can configure the monitoring component to monitor only Splittable WorkItems, Processable WorkItems, or both. In addition, the AllScale Monitor can also monitor WorkItems to a certain level of recursion.

3.3 Execution Reports

The AllScale Monitor Component can generate text reports at the end of program execution. These reports include raw measurements per WorkItem such as exclusive time, i.e. execution time of the WorkItem, as well as several derived metrics such as inclusive time per WorkItem, or mean execution time for all children of a WorkItem.

The AllScale Monitoring Component can also generate task dependency graphs, in other words, graphs with all the WorkItems executed by the runtime and their dependencies. WorkItem dependencies are created when a WorkItem is spawned from within another WorkItem due to the recursive model used. In our task graphs, each node represents a WorkItem and contains WorkItem name, exclusive execution time, and inclusive execution time. WorkItem graphs are coloured by exclusive time from yellow to red, being yellow shorter execution time and red longer execution time.

In addition, the monitoring framework can also generate heat maps with processes in the Y-axis and samples in the X-axis. These plots allow the user to inspect the evolution of certain metrics across processes and time, for example, the node idle rate, the WorkItem throughput, or the power consumed.

3.4 Introspection and Real-time Visualisation

The AllScale Monitor Component provides an API to access the collected performance data in real time, while the application runs. As previously explained, some performance measurements can be accessed directly whereas more complex metrics are calculated on-demand to reduce the overhead introduced into the application. Several of these on-demand metrics are computed recursively, taking advantage of the task dependency graph stored in memory, for example, the total time of a WorkItem and all its children, or the average execution time of the children for a certain WorkItem.

The AllScale Monitor Component provides a distributed global view of the performance, that is, there is an instance of the monitoring component per node, and each one of these instances can be directly asked for its performance data.

The introspection data can also be pipelined in real time via TCP to a web-server based dashboard, the AllScale Dashboard. This dashboard can be accessed with any available web browser.

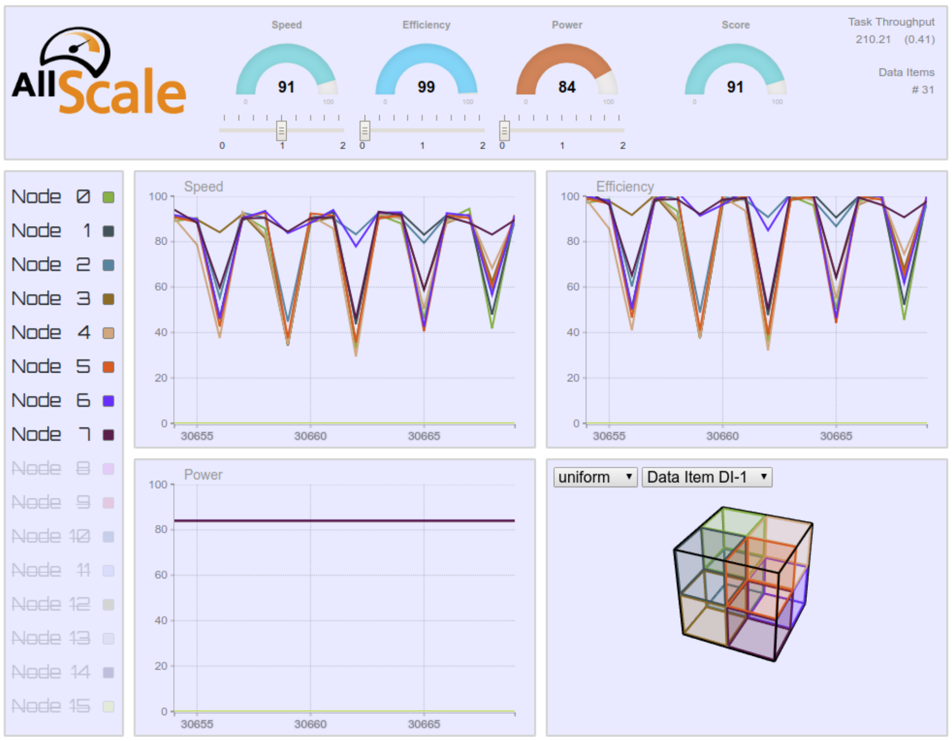
The AllScale Dashboard provides two views as shown in Fig. 1, a global view of the system, and a more detailed view per node. Fig. 1a shows the overview page of the system performance. In the upper part of the picture, the dashboard presents the system wide total speed, efficiency, and power for all nodes. The speed is the ratio of time doing useful work (executing WorkItems) against total time at maximum processor frequency. Efficiency is the ratio of useful time against the time at the current CPU speed and number of nodes used. Finally, power is the ratio of current power consumed against total power at the highest CPU frequency. Under each metric there is a slider bar that allows the user to change the weight of each objective in the multi-objective optimiser, thereby, being able to change the behaviour of the runtime in real time. The global score in the right part of the frame indicates how well the three customisable objectives (time, resource, power) are fulfilled. Finally, the frame also contains information of the system task throughput.

The global view also contains plots that depict the speed, efficiency, and power per node across time. Remember that this dashboard presents live information so all these plots evolve in real time while the application runs. In the right lower quadrant of the dashboard, we have a view on how the work is distributed across the system. Its dropdown menu allows the user to change in real time the policy used by the runtime to distribute such work. Thereby, allowing the user to observe how the work (and data) gets redistributed by changing the scheduling policy. The distribution of data within the runtime is unfortunately beyond the scope of this paper. This topic is covered in detail in [9].

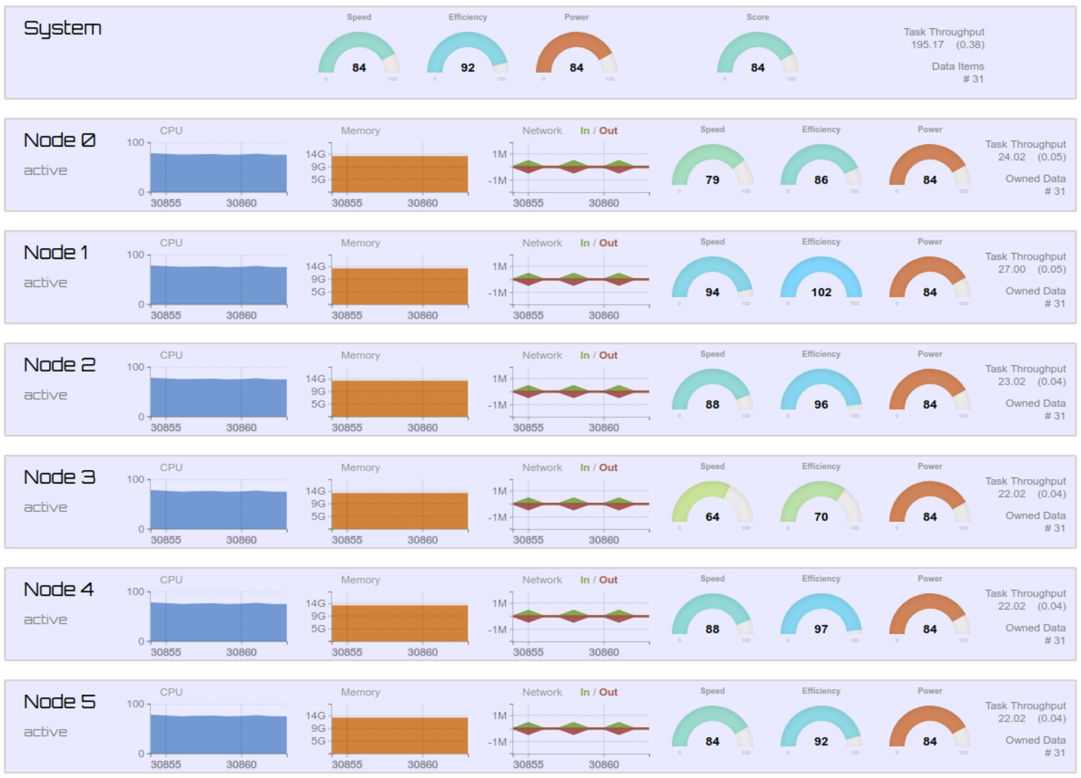
Fig. 1b shows the detailed view where performance data is depicted per node. For every node we have the CPU load, the memory consumption, the amount of data transferred through the network (TCP and MPI), speed, efficiency and power, the task throughput, and the amount of data owned by the node.

The AllScale Dashboard is a very powerful and convenient tool that can speed up the development process, because it allows developers to see as soon as they run the code the effect of source code changes. Furthermore, it also allows developers to change in real time the behaviour of the runtime to explore how the system behaves when changing different parameters.

The current version of the AllScale Dashboard uses one server that collects information from all the nodes involved in the execution. We are currently working on more scalable solutions to be able to tackle high node counts, for example, collecting the data in a tree manner, or having several hierarchically distributed servers. We also want to explore how to improve the graphical interface for high node counts, for instance, showing clusters of nodes instead of individual ones in the dashboard.

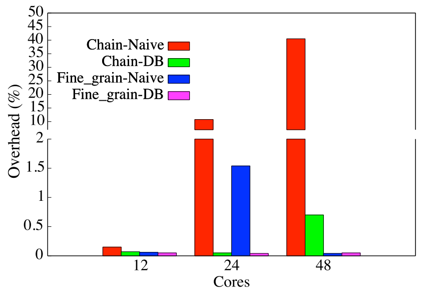


(a) Dashboard global system view.

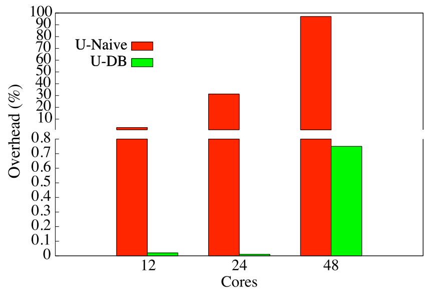


(b) Dashboard view with detailed metrics per node.

Fig. 1. The AllScale Dashboard showing performance data in real time while the application runs.



(a) Overhead with the stencil benchmarks



(b) Overhead with iPIC3D

Fig. 2. Overhead introduced by the AllScale Monitoring Component. Naive is an early implementation in which all threads process their own performance data. DB is the producer-consumer model with double buffering.

4 Evaluation

4.1 Overhead

The AllScale Monitoring Component has been designed with focus on having as minimum footprint in the system as possible to allow continuous performance introspection. We evaluated its overhead with several experiments. In the first set of experiments, we used a node with two 12-core E5-2690v3 Haswell and 512 GB of RAM memory, and run two benchmarks executing parallel loops that perform a simple stencil. The first benchmark (*chain* test) is a sequence of parallel loops where each loop waits for the completion of the previous one. The second benchmark (*fine_grain* test), on the other hand, computes loops in parallel by only waiting for the part of the previous loop it actually depends on. Both benchmarks were run with a grid of $128 \times 1024 \times 1024$ elements and for 100 time steps.

Figure 2a shows the overhead introduced by the monitoring framework into the stencil benchmarks running in 12, 24, and 48 logical cores (24 physical cores with hyper threading). Overhead is computed as the percentage difference in total execution time of the benchmark with and without the performance monitoring. In the experiments, the monitoring component collected execution time and dependencies for each WorkItem as well as node metrics every second. The picture depicts the overhead for two different versions of the monitoring component. First, an early naive implementation that we wrote where all threads process their own performance data and write it into per-process data structures. And second, our current producer-consumer implementation using a double buffering scheme.

As can be seen in the picture, the overhead increases as we increase the number of cores, especially in the naive implementation due to its high thread contention. It is particularly bad for the chain benchmark, going from less than

0.5% to around 40%. In contrast, the overhead introduced when using our double buffering implementation is much lower, being always less than 1%. We can also see in the plot that there is a difference in the overhead between both benchmarks. This difference is caused by the nature of each benchmark. While the fine_grain benchmark executes all the iterations in parallel, the chain benchmark imposes threads to wait for the previous iteration to be finished. This synchronisation prevents the updates to the shared information to spread out, increasing the contention in global data structures.

We also evaluated the overhead of the monitoring component using a real-world application: iPIC3D [13], a Particle-in-Cell code for space plasma simulations that simulates the interaction of Solar Wind and Solar Storms with the Earth's Magnetosphere. In our evaluation we used sets of 8×10^6 particles uniformly distributed, and run 100 time steps of the simulation.

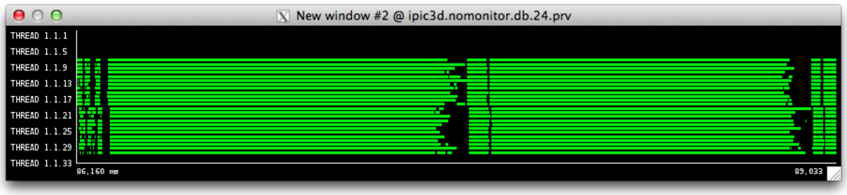
Figure 2b shows the overhead introduced by both implementations of the AllScale Monitoring Component. As can be seen in the picture, the naive implementation has a high overhead that increases rapidly with the number of threads used. This naive implementation introduces more than 90% of overhead with 48 logical cores. Thus, demonstrating that classical implementations where each thread processes its own data do not work in a scenario where lots of threads execute a huge amount of small tasks. For the double buffering implementation, the overhead is always smaller than 1%.

Figure 3 shows three different Paraver timelines of iPIC3D where we can see how the monitoring infrastructure interacts with the application. Figure 3a depicts two time steps of the application with the AllScale Monitoring turned off. We can see in Fig. 3b how our current AllScale Monitor Component implementation does not affect the application. In contrast, figures 3c serves to expose how the first naive approach we implemented affects the application execution a lot.

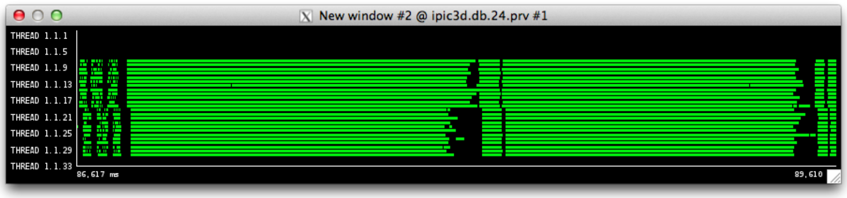
We also performed distributed experiments in a Cray XC40 equipped with two Intel Xeon E5-2630v4 Broadwell at 2.2 GHz and 64GB of RAM per node. Nodes were connected with an Intel OmniPath interconnect. Fig. 4 shows the running time of iPIC3D for different number of particles per node up to 64 nodes. The graph contains for each number of particles the simulation time with and without the monitoring infrastructure. The monitoring component used is the producer-consumer version with double buffering. As can be seen in the picture, the overhead of our monitoring component is negligible. It can also be observed that in some cases the execution time with the monitoring component activated is shorter than without monitoring. This can be explained by the fact that the overhead is so small that it gets absorbed by the natural execution noise and job execution time variability present in HPC systems.

4.2 Use Case

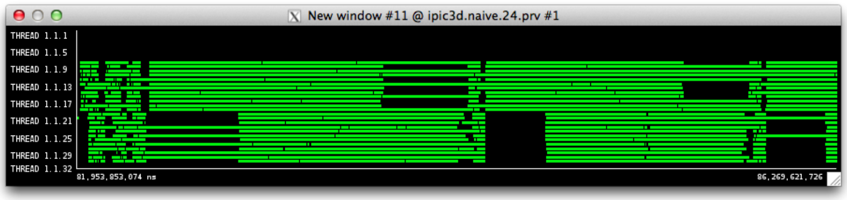
As previously explained, the AllScale Runtime can optimise the execution of applications regarding three different weighted objectives: time, resource, and energy. The runtime contains a prototype of an optimiser module that implements



(a) WorkItem execution in iPIC3D without the AllScale Monitoring (24 cores)



(b) WorkItem execution in iPIC3D with the AllScale Monitoring using double buffering (24 cores)



(c) WorkItem execution in iPIC3D with the naive implementation of the AllScale Monitoring (24 cores)

Fig. 3. Paraver timelines for two iterations of iPIC3D. Y-axis contains threads and X-axis time. Green is WorkItem execution. Black is outside WorkItem execution.

the Nelder-Mead method [14] to search for the best set of parameters that fulfil the objectives set. There are several parameters that the runtime can tune while trying to fulfil such objectives, for instance, number of threads and frequency of the processors used. The multi-objective optimiser uses the performance data provided by the AllScale Monitor Component to guide this parameter optimisation. It uses power consumed, system idle rate and task throughput among others.

In this section, we demonstrate how the AllScale Runtime uses in real time the introspection data collected by the AllScale Monitor Component to run an application as energy efficient as possible. To that aim, we run iPIC3D with three different tests cases: one with a uniformly distributed set of particles in a nebula (uniform case), one with a non-uniformly statically distributed set of particles (cluster case), and finally, one with a non-uniformly dynamically changing set

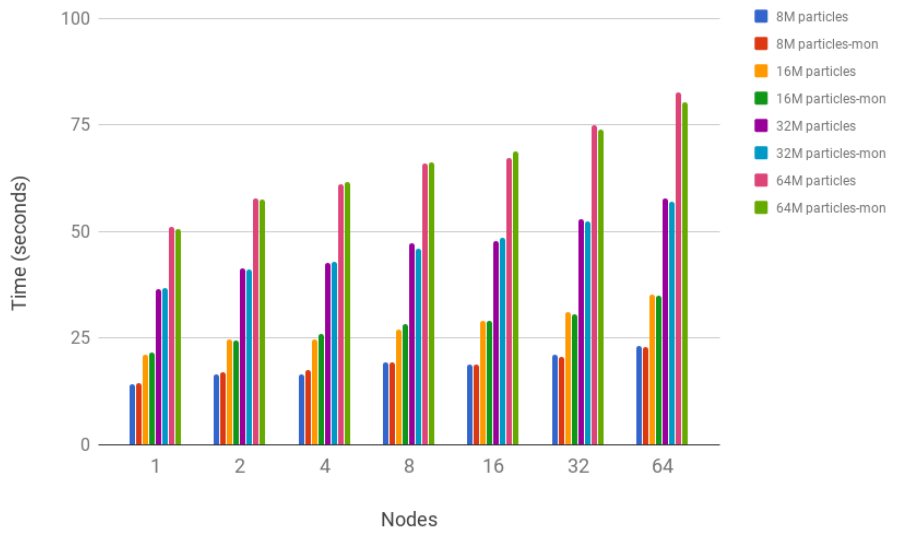
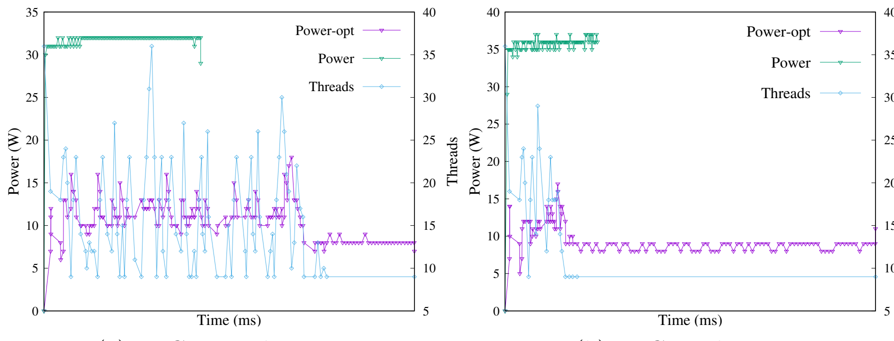


Fig. 4. Execution times for different configurations of particles per node with and without the AllScale Monitor Component. Columns with the suffix -mon are the runs with the monitoring infrastructure activated.

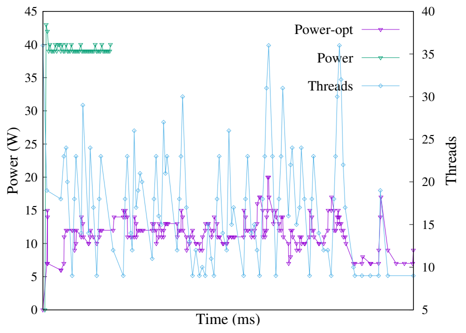
of particles in an explosion (explosion case). We run the experiments in a node with 2 Intel Xeon Gold processors (36 cores in total), and 16×10^6 particles.

In the experiments we gave the maximum weight to the energy objective, that is, the runtime will try to optimise only by the energy consumed. To this end, the runtime will dinamically tune the number of threads and the frequency of the processors used. Fig. 5 shows the power consumed for the three test cases while running with and without the optimiser. The green line (label *Power*) shows the power when running without any self-tuning using the *ondemand* governor for the CPU frequencies. With the *ondemand* governor the operating system manages the CPU frequencies. The purple line (label *Power-opt*) represents the power consumed when using the optimiser with the energy objective. The blue line shows the number of threads the runtime is using when the optimiser is tuning the run.

We can see several things in the figure. First, we can see how the optimiser starts always with the maximum number of threads and changes it until it finds an optimal value. Second, it is noticeable that even though the runs without optimisation (green line) consume much more power, they are much shorter. In terms of total energy consumed, in the explosion case, the version using the optimiser consumes a total of 4,066.97 J compared to 5,400.48 J without it. For the cluster case the numbers are 3,758.3 J with the optimiser against 3,696.75 J without it. Finally, for the uniform case, the run with the optimiser consumes 3,197.12 J against 2,064.93 J without it. As can be seen from the numbers, the



(a) iPIC3D explosion case (b) iPIC3D cluster case



(c) iPIC3D uniform case

Fig. 5. Power consumed by iPIC3D when running with (power-opt line) and without (power line) the AllScale Multi-objective Optimiser.

current optimiser prototype does not find an optimal configuration for the cluster and uniform cases. Fig. 5c for example, shows how the optimiser does not find an optimal thread number and keeps changing it during the whole execution. Thus, further investigation on why the current optimiser prototype does not find optimal solutions for these two test cases is needed. It is important to remark however that thanks to the AllScale Monitor Component we have been able to detect the bad behaviour of the optimiser module, thereby, being able to warn their developers and helping them to improve it.

5 Related Work

Performance observation and analysis is a topic extensively explored in HPC. There are many tools that can be used for post-mortem performance analysis of parallel applications, e.g. Paraver [15], Score-P [12], Scalasca [6], or TAU [17]. These tools are really useful to detect and optimise performance problems, however, they were designed for post-mortem analysis and are not sufficient for continuous monitoring introspection. In contrast, our monitoring component is not a stand-alone tool but a runtime system component, which cannot provide the same level of performance detail as the previous tools, but allows continuous monitoring introspection with very low overhead.

Most state-of-the-art runtime systems include means to generate performance reports for post-mortem analysis. StarPU [2] uses the FXT library [4] to generate Pajé traces and graphs of tasks. Legion [3] provides profiles and execution timelines with performance data of the tasks executed by the runtime. OmpsSs [5] generates Paraver [15] traces with Extrae. It also provides task dependency graphs in pdf format. APEX [7] generates performance profiles that can be visualised with TAU [17].

Nevertheless, classical post-mortem techniques will not suffice in the Exascale era. Real-time introspection capabilities are a requirement for performance tuning and adaptation. Thus, several performance tools and task-based runtimes implement introspection strategies. Tools such as the work of Aguilar et al. [1] or Score-P [12] include online introspection capabilities. However, these tools are mainly designed for MPI monitoring. The runtime Legion [3] contains task mappers that can request performance information to the runtime while the application runs. OmpSs has an extension to use introspection data to guide the process of task multi-versioning [16]. StarPU provides an API to access performance counters from the runtime.

6 Conclusions and Future Work

The path to Exascale brings new challenges to scalability, programmability, and performance of software. Future systems will have to include performance introspection to support adaptivity and efficient management of resources. In this paper we have presented the AllScale Monitor Component, the monitoring infrastructure included within the AllScale Runtime. This monitoring framework provides real-time introspection to the AllScale Scheduler with minimum overhead as we have demonstrated. We have also shown that real-time introspection capabilities are very useful to orchestrate application execution as well as to analyse the performance of the system in real time. Thus, speeding up the development process because performance deficiencies can be detected almost instantly, without the need to wait until the end of a long running job for example. In addition, the monitoring framework is self-contained, so it can be easily decoupled and used in other runtimes.

As future work, we want to further investigate the use of historical performance data together with real-time introspection data to help in the decision-making process of the scheduler. We are also working in improving the scalability of the dashboard server as well as the scalability of its graphical interface.

References

1. Aguilar, X., Laure, E., Furlinger, K.: Online performance introspection with IPM. In: High Performance Computing and Communications & 2013 IEEE International

Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on. pp. 728–734. IEEE (2013)
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2), 187–198 (2011)
3. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: Proceedings of the international conference on high performance computing, networking, storage and analysis. p. 66. IEEE Computer Society Press (2012)
4. Danjean, V., Namyst, R., Wacrenier, P.A.: An efficient multi-level trace toolkit for multi-threaded applications. In: European Conference on Parallel Processing. pp. 166–175. Springer (2005)
5. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters **21**(02), 173–193 (2011)
6. Geimer, M., Wolf, F., Wylie, B.J., Ábrahám, E., Becker, D., Mohr, B.: The scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience **22**(6), 702–719 (2010)
7. Huck, K.A., Porterfield, A., Chaimov, N., Kaiser, H., Malony, A.D., Sterling, T., Fowler, R.: An autonomic performance environment for exascale. Supercomputing frontiers and innovations **2**(3), 49–66 (2015)
8. Jordan, H.: Insieme: A compiler infrastructure for parallel programs. Ph.D. thesis, Ph. D. dissertation, University of Innsbruck (2014)
9. Jordan, H., Heller, T., Gschwandtner, P., Zangerl, P., Thoman, P., Fey, D., Fahringer, T.: The Allscale Runtime Application Model. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER). pp. 445–455. IEEE (2018)
10. Jordan, H., Thoman, P., Zangerl, P., Heller, T., Fahringer, T.: A Context-Aware Primitive for Nested Recursive Parallelism, pp. 149–161. Springer International Publishing (2017)
11. Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: Hpx: A task based programming model in a global address space. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. p. 6. ACM (2014)
12. Knüpfer, A., Rössel, C., Mey, D.a., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir, pp. 79–91. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
13. Markidis, S., Lapenta, G., et al.: Multi-scale simulations of plasma with iPIC3D. Mathematics and Computers in Simulation **80**(7), 1509–1519 (2010)
14. Nelder, J.A., Mead, R.: A simplex method for function minimization. The computer journal **7**(4), 308–313 (1965)
15. Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: Transputer and occam Developments. vol. 44, pp. 17–31 (1995)
16. Planas, J., Badia, R.M., Ayguade, E., Labarta, J.: Self-adaptive ompss tasks in heterogeneous environments. In: Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. pp. 138–149. IEEE (2013)
17. Shende, S.S., Malony, A.D.: The TAU parallel performance system. International Journal of High Performance Computing Applications **20**(2), 287–311 (2006)