TINS: A Task-Based Dynamic Helper Core Strategy for In Situ Analytics

Estelle Dirand[1(✉)], Laurent Colombet[1], and Bruno Raffin[2]

[1] CEA, DAM, DIF, 91297 Arpajon, France
estelle.dirand@cea.fr
[2] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

Abstract. The in situ paradigm proposes to co-locate simulation and analytics on the same compute node to analyze data while still resident in the compute node memory, hence reducing the need for post-processing methods. A standard approach that proved efficient for sharing resources on each node consists in running the analytics processes on a set of dedicated cores, called helper cores, to isolate them from the simulation processes. Simulation and analytics thus run concurrently with limited interference. In this paper we show that the performance can be improved through a *dynamic helper core strategy*. We rely on a work stealing scheduler to implement TINS, a task-based in situ framework with an on-demand analytics isolation. The helper cores are dedicated to analytics only when analytics tasks are available. Otherwise the helper cores join the other cores for processing simulation tasks. TINS relies on the Intel® TBB library. Experiments on up to 14,336 cores run a set of representative analytics parallelized with TBB coupled with the hybrid MPI+TBB ExaStamp molecular dynamics code. TINS shows up to 40% performance improvement over various other approaches including the standard helper core.

1 Introduction

The exascale era will bring more computational capabilities enabling the simulation of more complex phenomena with higher precision. This will generate a growing amount of data. Traditionally, simulation codes output data into the filesystem and these data are later read back for postmortem analytics. However, the growing gap between computational capabilities and IO bandwidth calls for new data processing methods.

The *in situ* paradigm proposes to reduce data movement and to analyze data while still resident in the memory of the compute node by co-locating simulation and analytics on the same compute node [1]. The simplest approach consists in modifying the simulation timeloop to directly call analytics routines. However, several works have shown that an *asynchronous* approach where analytics and simulation run concurrently can lead to a significantly better performance [2–4]. Today, the most efficient approach consists in running the analytics processes on

R. Yokota and W. Wu (Eds.): SCFA 2018, LNCS 10776, pp. 159–178, 2018.
https://doi.org/10.1007/978-3-319-69953-0_10

a set of dedicated cores, called helper cores, to isolate them from the simulation processes [3]. Simulation and analytics thus run concurrently on different cores but this static isolation can lead to underused resources if the simulation or the analytics do not fully use all the assigned cores.

In this paper, we introduce TINS, a task-based in situ framework that implements a novel *dynamic helper core* strategy. TINS relies on a work stealing scheduler and on task-based programming. Simulation and analytics tasks are created concurrently and scheduled on a set of worker threads created by a single instance of the work stealing scheduler. Helper cores are assigned dynamically: some worker threads are dedicated to analytics when analytics tasks are available while they join the other threads for processing simulation tasks otherwise, leading to a better resource usage. We leverage the good compositionality properties of task-based programming to seamlessly keep the analytics and simulation codes well separated and a plugin system enables to develop parallel analytics codes outside of the simulation code.

TINS is implemented with the Intel® Threading Building Blocks (TBB) library that provides a task-based programming model and a work stealing scheduler. The experiments are conducted with the hybrid MPI+TBB ExaStamp molecular dynamics code [5] that we associate with a set of analytics representative of computational physics algorithms. We show up to 40% performance improvement over various other approaches, including the standard helper core, on experiments on up to 14,336 Broadwell cores.

The paper is organized as follows. After an overview of related work (Sect. 2), we present the TINS task-based in situ method (Sect. 3) and we compare the dynamic helper core method with state-of-the art approaches (Sect. 4).

2 Related Work

The more direct way to perform in situ processing is called *synchronous* and consists in in-lining analytics code in the simulation code. The total execution time is the addition of simulation and analytics times, plus some possible overheads due to cache trashing. The analytics can directly access the simulation data structures, but more often a copy is performed to build a data structure adapted to the analytics needs [6]. ParaView/Catalyst [7] and VisIt/Libsim [8] are both relying on this approach to enable in situ visualization. They recently worked on a unified in situ API for the simulation codes, called SENSEI [9], to switch between Catalyst, Libsim and the IO framework ADIOS [10] with very limited code modifications.

Parallel simulations are almost never 100% efficient, some cores being idle during communication phases for instance or because some code sections do not provide enough parallelism to feed all the cores. One idea is to harvest these CPU cycles to execute analytics, leading to execution times shorter than with the synchronous execution. This is called *asynchronous in situ*. A simple approach consists in relying on the OS scheduler capabilities to allocate resources. The analytics run its own processes or threads concurrently with the ones of the

simulation. The simulation only needs to give a copy of the relevant data to the local in situ analytics processes. The analytics can next proceed concurrently with the simulation. However, works [11,12] show that relying on the OS scheduler does not prove efficient because the presence of analytics processes tends to disturb the simulation.

To circumvent this problem, a common approach consists in dedicating one or more cores, called *helper cores*, to the analytics. The simulation runs on less cores, but, because it is usually not 100% efficient, its performance decreases by less than the ratio of confiscated cores. Damaris [3], FlowVR [2] Functional Partitioning [13], GePSeA [14], Active Buffer [15] or FlexIO [4] support this approach and have demonstrated its benefit in different contexts. Performance gains are usually significant compared to a synchronous approach. However, because the analytics and simulation are both isolated on distinct subsets of cores, this helper core strategy does not allow the analytics to harvest unused cycles of the simulation cores and vice versa.

GoldRush [11] takes a different approach. It implements a custom time-sharing scheduling to interleave simulation and analytics while limiting the interference on the simulation. Goldrush detects sequential sections in the OpenMP code of the simulation to schedule the analytics processes. The simulation sends resume signals to the analytics during these sections while the analytics are suspended otherwise. Experiments show the simulation performance is improved compared to OS controlled scheduling or a synchronous approach. However, Goldrush does not enable overlapping simulation and analytics during short simulation sequential sections and weakly scalable parallel sections.

All previously mentioned approaches applied to MPI or MPI+OpenMP simulations. New programming models are also developed as alternatives to message passing. StarPU [16], PaRSEC [17], Legion [18] and HPX [19] propose task-based runtime systems for distributed heterogeneous architectures. The program defines a directed acyclic graph where vertices are tasks and edges data dependencies between tasks. The runtime is in charge of mapping tasks to resources, and triggering task execution and the necessary data movements when data dependencies are resolved. Early experiments have been reported using Legion for in situ analytics [20,21]. They show that Legion runtime is able to overlap analytics and simulation tasks, but globally the performance is not yet competitive with MPI approaches.

In a more general context the shortcomings of standard OS for scheduling concurrent parallel applications on one multi-core node motivated the development of specific *co-scheduling* strategies. Space-sharing is often favored compared to time-sharing as it usually leads to better performance. But these solutions require a specific OS scheduler or modifications to the parallel runtimes [12,22].

3 The TINS Framework

3.1 Work Stealing and TBB

Task-based programming is becoming a standard for shared memory. The user only needs to delimit the potential parallelism through tasks or loops and the runtime takes care of creating and distributing these tasks to the worker threads it created. In a *work stealing scheduling*, the threads are assigned a set of tasks they have to execute. When a thread has executed all its tasks, it selects another thread and steals part of this victim's tasks if available; otherwise, it tries with another victim. The work stealing scheduler algorithm has a proven performance [23]. Pioneered by Cilk, task-based programming is today also available through Intel® TBB or OpenMP for instance.

In this paper, we use the TBB library that provides a task-based programming model and a work stealing scheduler for shared memory machines. TBB provides mechanisms to guide the task execution, in particular the notions of `task_arena` (arena in the following) and `task_scheduler_observer` (observer in the following). An *arena* encapsulates one or several TBB parallel regions where threads share and execute tasks. An arena is defined with a *concurrency level* that fixes the maximum number of tasks that can be executed simultaneously. In other words, the arena concurrency level determines the maximum number of threads that can work inside an arena. An application can contain several arenas. In this situation, when the parallel work encapsulated in an arena has been completed, the worker threads involved in this arena are free to enter another arena if its concurrency level allows it. An *observer* is an object that intercepts when a worker thread enters and leaves a specific arena. We use it to control thread affinity as detailed in Sect. 3.4.

In a TBB application, there will never be more threads running than the number of cores in the processor to avoid core oversubscription. In the case of an application with two concurrent arenas with concurrency levels of n_1 and n_2 on a processor with N cores, two situations can therefore be distinguished:

- if $n_1 + n_2 \leq N$, the concurrent arenas can have as many threads as requested (there will be n_1 threads in the first arena, n_2 in the second);
- if $n_1 + n_2 > N$, the concurrent arenas cannot have as many threads as requested and TBB allocates to each arena a number of threads proportional to the request ($n_1/(n_1 + n_2)N$ and $n_2/(n_1 + n_2)N$ respectively).

3.2 In Situ Processing with Tasks

TINS relies on the TBB work-stealing scheduler to implement a novel task-based in situ processing method. Simulation and analytics tasks are created concurrently and scheduled on a set of worker threads created by a single instance of the TBB scheduler. The use of TBB arenas allows to implement two asynchronous patterns: the standard helper core strategy with a permanent thread isolation and a *dynamic helper core strategy* with a temporary thread isolation.

3.3 Spawning Analytics and Simulation Tasks

Traditionally, a simulation is organized around a timeloop where internal data are updated at each timestep. We consider here a hybrid MPI+TBB simulation where each MPI process runs one instance of the work-stealing scheduler. Following TBB vocabulary, we call *simulation master thread* the simulation main thread started by MPI for each process. When tasks are created, they are distributed among the simulation master thread and the *worker threads* spawned by TBB.

To enable the asynchronous execution of the analytics, we propose the method described in Fig. 1. The simulation master thread spawns an *analytics master thread* at simulation initialization. The simulation and analytics master threads have their own timeloop and arena with different concurrency levels: the simulation master thread creates simulation tasks in the simulation arena while the analytics master thread creates analytics tasks in the analytics arena. Each master thread is responsible for its own arena and cannot enter the other one, while worker threads can change of arena as detailed in Sect. 3.4.

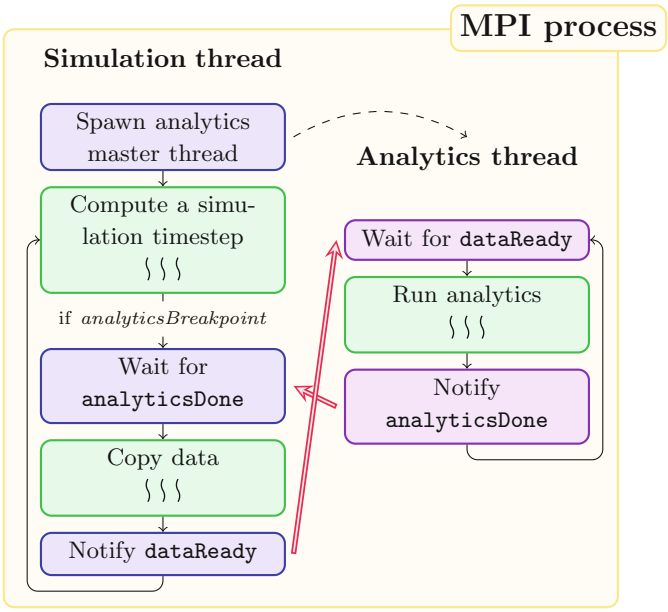

Fig. 1. Timeloops of the simulation (left) and analytics (right) master threads inside one MPI process. The green-framed blocks contain sequential regions (MPI communications for example) and parallel regions where simulation or analytics tasks are scheduled on the worker threads spawned by TBB inside the MPI process. The red arrows depict the synchronization between the master threads. (Color figure online)

The computation of the simulation timestep is left unchanged by TINS, alternating sequential regions with parallel ones where simulation tasks are created.

The user defines an *analytics breakpoint frequency* that sets the frequency of data processing. Every time the simulation reaches such analytics breakpoint, data are copied into a temporary buffer. When data are copied, the simulation master thread notifies the analytics master thread that data are ready to be processed with the `dataReady` signal and resumes the simulation execution.

On the other side, the analytics master thread waits for the simulation master thread `dataReady` signal to launch the analytics on the data written into the temporary buffer. It creates analytics tasks while the simulation master thread creates simulation tasks in its own timeloop, leading to an asynchronous in situ pattern. Once the analytics are executed, the analytics master thread notifies the simulation master thread with the `analyticsDone` signal. This second synchronization is added to avoid having to store more than one temporary buffer. This synchronization can be delayed if enough memory is available to store various buffers. The simulation master thread therefore has to wait for the `analyticsDone` signal before writing data in the temporary buffer. This signal is disabled for the first analytics breakpoint to avoid a deadlock.

3.4 Resource Sharing Policies

Analytics tasks can be executed in the two asynchronous modes described in Fig. 2. To do so, we define two arenas with concurrency levels n_s and n_a for simulation and analytics respectively. In order to simply manage the arenas and the asynchronous modes, we defined two functions that need to be placed before and after the TBB parallel regions.

On a processor with N cores, TBB spawns up to $N - 1$ worker threads by default, which would lead to core oversubscription because there are already 2 master threads. To avoid this pitfall, we pin the analytics master thread on the first core thanks to the TBB observer and we restrict the node topology so that the TBB scheduler only sees the remaining $N - 1$ cores. This way, there will be at most $N - 2$ worker threads. Various pinning strategies were tested on the simulation master thread. Because no solution outperformed the other, we decided not to pin it. The placement of the worker threads depends on the strategy.

In the *static helper core* strategy, the available threads are split in two categories: some threads execute analytics tasks while the other ones are in charge of simulation tasks. The isolation is permanent. In particular, threads remain idle when no task of the expected kind is available for execution. To implement the static helper core strategy, the concurrency levels are chosen such that $n_a + n_s = N$. The TBB observer is used to bind threads that execute analytics tasks on the first n_a cores of the processor while the threads that execute simulation tasks are bound to the remaining cores. The goal is to try as much as possible to gather all threads of the same kind on the same NUMA nodes for a better cache efficiency. Tests showed that it notably improves the performance.

We introduce the *dynamic helper core* policy with a temporary thread isolation. As in the static helper core approach, a set of threads is assigned to analytics

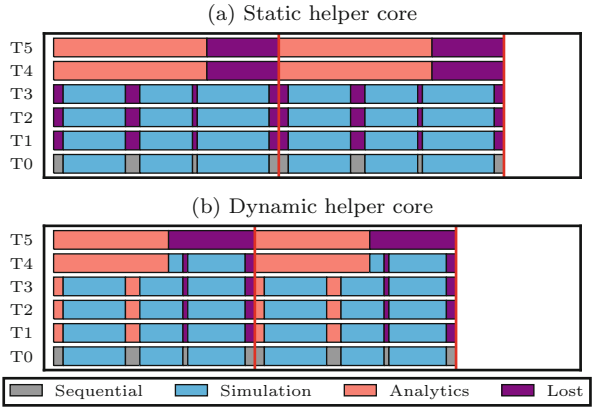

Fig. 2. Gantt diagram of the execution of simulation and analytics tasks on 6 threads (T0 to T5) for a static (a) or dynamic (b) helper core strategy. T0 and T5 are respectively the simulation and analytics master threads, T1, T2 and T3 are worker threads assigned to simulation and T4 is a worker thread assigned to analytics. The diagram represents two iterations of a simulation, both being the alternation of four sequential regions (grey areas) and three parallel regions (blue areas). The analytics is composed of one parallel region (orange areas). The purple areas highlight the periods when the threads are idle. The dynamic helper core strategy enables worker threads to switch to simulation (resp. analytics) tasks when there is no analytics (resp. simulation) work left, while this is not possible with static helper cores. (Color figure online)

tasks execution while the remaining execute simulation tasks. The main difference with the static approach is that the isolation is temporary: when the execution of a simulation (resp. analytics) parallel region is completed, the worker threads involved in the computation can enter the analytics (resp. simulation) arena if its concurrency level permits it. This method aims at reducing the thread idleness periods induced by the static helper core approach. We set $n_s = N - 1$ so that all the worker threads and the simulation master thread can work on simulation tasks if available. Note that the analytics master thread cannot execute simulation tasks because it is not allowed to enter the simulation arena. To restrict the number of threads in the analytics arena, we can choose different values for n_a. $n_a = n_s$ means that half of the threads will execute analytics tasks when both arenas are active while $n_a < n_s$ gives a higher priority to the simulation. We tested various binding strategies for the worker threads, but because they can execute tasks from both arenas, we did not observe that a binding strategy was overcoming the others. We therefore adopted the less constraining one by not binding the worker threads.

3.5 Plugin System

TINS aims at keeping the simulation and analytics codes well separated. We therefore developed a plugin system that allows to develop the analytics outside

of the simulation code. A plugin is a code compiled as a shared library. At runtime, the analytics master thread scans the plugin directory provided by the user and loads the required analytics. This way, simulation and analytics tasks are scheduled by the same instance of the TBB scheduler. A plugin should meet the following requirements. First, it has to be developed using a MPI+TBB programming model and it should take as input a MPI communicator. Indeed, simulation and analytics may perform MPI communications simultaneously and they need to use distinct communicators for the messages not to be mixed. The analytics master thread therefore creates its own communicator that the plugins should use for their internal MPI communications. To ease the interoperability between the simulation code and the plugins, a shared data structure also needs to be defined and used by both the simulation and the plugin. The simulation copies the data in this shared data structure and the plugin takes it as input.

4 Experimental Evaluation

We compare the dynamic helper core strategy implemented with TINS with several other approaches on a molecular dynamics simulation using Intel® Xeon processors available in the CCRT French Computing Center.

4.1 ExaStamp Molecular Dynamics Code

ExaStamp [5] is a molecular dynamics code dedicated to material sciences (condensed matter and shock physics). It is written in C++11 and uses MPI and TBB for the different levels of parallelism. ExaStamp is used as a production code and routinely runs on more than 4,000 cores to simulate the displacement of up to 1 billion particles in a 3D system. ExaStamp is well parallelized, leaving limited compute resources under-used: it shows an efficiency of 90% on one node varying the number of cores from 1 to 28, and of 85% when scaling from 1 to 512 nodes. For each particle, the parameters of interest are the index of the particle (`idx`), its type (`type`), its position along the three axes (`rx`, `ry`, `rz`) and its velocity along the three axes (`vx`, `vy`, `vz`). To ease the interoperability between ExaStamp and the analytics described below, we defined the `ParticleInSitu` data structure as a structure of arrays where each array contains *nbPart* elements, *nbPart* being the number of particles in the current MPI process. The data structure is shared by the simulation code and the analytics implemented in the plugin system: the simulation produces and fills it and the plugins take it as input.

Implementing the TINS approach in ExaStamp required about 50 extra lines of code. The analytics master thread is implemented as a C++ thread and the synchronization signals between the master threads are implemented with shared booleans. The master threads may perform MPI communications concurrently so we need a thread-safe implementation of MPI with `MPI_THREAD_MULTIPLE`.

ExaStamp execution is parametrized through an input data file that defines the analytics to be executed, the analytics breakpoint frequency, the execution policy and the size of the arenas. No recompilation is required to change the configuration.

4.2 Analytics

To test TINS, we developed a set of analytics routines representative of the analytics used in computational physics (Table 1). They were chosen to represent different patterns regarding parallelization, MPI communications, cache and memory usage.

Table 1. Analytics implemented to evaluate TINS

Analytics	Description
`write_dat`	Write the positions of the particles inside each MPI process in a file (one file per MPI process)
`statistics_seq`	Compute sequentially the mean of the positions for the particles inside each MPI process
`statistics_par`	Compute in parallel the mean of the positions for the particles inside each MPI process (with 1 TBB parallel reduction)
`radial`	Compute in parallel a local radial distribution function for the particles inside each MPI process (with 2 nested TBB parallel for)
`histogram`	Compute in parallel a global histogram of `rx` positions (locally with 2 TBB parallel reductions, and globally with 2 `MPI_REDUCE`)

In the `write_dat` routine, each MPI process writes a file with the positions of each particle at each analytics breakpoint. This analytics mimics a native file writing pattern commonly used in ExaStamp to write particles in an XYZ format suitable for post-processing tools. This analytics plugin neither generates TBB tasks nor MPI communications.

The two statistics routines trigger local computations and do not perform any MPI communication. They both compute the mean of the positions of the particles from the data copied in each MPI process. We implemented a sequential version (`statistics_seq`) and a parallel version (`statistics_par`) where the mean is computed through one TBB parallel reduction. Each task consists in a few summations but is very memory intensive. When simulating the behavior of 4,000,000 particles per MPI process, the positions represent approximately 96 MB of data per MPI process, significantly more than the caches available on a Broadwell processor (see below for the processor specifications). Reading these data therefore evicts simulation data from the caches. Moreover, these analytics highlight NUMA effects because data are split between the caches of the different NUMA nodes. To further stress memory accesses for the experiments, the statistics routines can be executed several times at each analytics breakpoint.

The histogram algorithm (`histogram`) mixes TBB tasks creation and MPI communications. This routine counts how many particles have a position in intervals of the form $[rx_i, rx_i + \Delta x]$. A first collective communication is necessary to determine the bounds of the system: each MPI process computes its own minimum and maximum positions with a TBB parallel reduction and the global

bounds are found thanks to a `MPI_REDUCE` operation. The global domain is then split into smaller intervals of the form $[rx_i, rx_i + \Delta x]$. The number of particles in each interval is computed inside each MPI process thanks to a TBB parallel reduction and the global histogram is then computed with a `MPI_REDUCE`. The histogram is computed on 1,000 intervals. For experimenting with analytics having different MPI communication loads, we can increase the size of the arrays communicated in the second `MPI_REDUCE`. This way, we can see the influence of an analytics that spends most of its time in MPI communications.

The local radial distribution function (`radial`) is a common algorithm in computational physics and consists in a local histogram over the distances between the particles. For each particle, we compute the distance with all the other particles and store them in a local histogram of 1,000 bins. This analytics requires two nested for loops and is parallelized with TBB thanks to the `tbb::blocked_range2d` feature. This algorithm is used because it demonstrates the effect of a compute intensive analytics.

4.3 I/O Middlewares

We compare the TINS approach with two state-of-the-art in situ frameworks: Damaris [24] and Goldrush [11].

Damaris implements the static helper core strategy. It is a MPI-based approach that starts on each node a certain number of processes for the simulation and the analytics, each one running with their own MPI communicator. Local data transfers from the simulation to the analytics processes are made through a shared memory segment. To limit data copies, Damaris enables the simulation to directly allocate data inside the shared memory segment. The simulation writes data into this shared memory segment and the analytics deallocates data once consumed. We instrumented ExaStamp with the Damaris API and developed Damaris plugins for the five analytics described above, keeping their TBB parallelization when existing. An important difference with TINS is that Damaris starts two distinct instances of TBB scheduler per node: one for running the simulation tasks and the other for the analytics. Damaris does not integrate mechanisms for pinning the processes or threads to the cores. We use TBB observers to bind analytics threads (master and workers) to the helper cores and the simulation threads (master and workers) to the remaining cores. The helper cores are assigned contiguously starting from the first core to keep them running as much as possible on the same sockets. We experienced better performance with this approach.

Goldrush is a C library that implements a custom time-sharing scheduling to trigger analytics during the simulation sequential sections. Each simulation process records the duration of its sequential sections and assumes these sections repeat at each iteration. When a sequential section is long enough, given a user-defined threshold, the simulation process sends a SIGCONT signal to resume

the analytics process and a SIGSTOP signal to suspend it at the end of this sequential region. We instrumented ExaStamp with the Goldrush API delimiting the sequential regions where no TBB task is created. We ported the sequential statistics and the parallel one with its TBB parallelization that can run tasks on all cores when resumed by Goldrush.

4.4 Experimental Setups

Experiments run on the Cobalt supercomputer from the CCRT high performance computing center. Each node has two Intel® Broadwell CPUs running at 2.40 GHz and 128 GB of memory. Each CPU has 2 NUMA nodes with 7 cores each and a shared L3 cache of 17,920 KB. Hyperthreading is not activated. The nodes are connected through a EDR InfiniBand network. The codes use Intel® TBB 16.0.3.210, are compiled with `icpc` compiler (version 17.0.4.196) and are launched with Intel® MPI (version 2017.0.4.196).

Experiments are conducted timing 32 consecutive iterations of ExaStamp, with the analytics performed after each timestep. In production codes, outputs are usually not produced at each timestep to avoid slowing down to much the execution. Here we stress the system by analyzing data at each iteration to make the overheads more visible.

Tests are performed on simulations with 4,000,000 particles per MPI process and one MPI process per node. Simulation codes usually run several MPI processes per node, but we run only one MPI process per node to probe TBB scheduler with a larger pool of cores. We compared the performance of running ExaStamp with 1 process per node and 4 processes per node and measured only a 2% performance drop.

4.5 Results

Comparison with Goldrush

We first compare the TINS approach with the static and dynamic helper core strategies with the Goldrush approach. We ran three analytics on 28 Broadwell cores for a simulation of 4,000,000 particles: the parallel statistics performed 100 and 1,000 times at each analytics breakpoint (`stat_par_100` and `stat_par_1000`) for small and long analytics parallelized with TBB and the sequential statistics computed 1,000 times at each analytics (`stat_seq_1000`) for a long analytics without parallelization. For each experiment, we tested two static helper core configurations (`SHC-a-s`) and two dynamic helper core configurations (`DHC-a-s`) where `a` and `s` stand for analytics and simulation arena sizes.

Table 2 shows that the Goldrush approach is efficient on small parallel analytics. For instance, it gives on overhead of only 8% on ExaStamp executed without analytics (`ExaStamp-alone`) when co-locating the `stat_par_100` analytics and it can outperform the TINS approach with 7 static helper cores because too much cores were removed from the simulation in this situation. However, the TINS approach with dynamic helper core strategy can be up to 4.55% faster than the

Table 2. Total execution times in seconds of ExaStamp co-located with three analytics executed with different TINS configurations and with Goldrush for a simulation of 4,000,000 atoms on 28 Broadwell cores (1 MPI process)

	`stat_par_100`	`stat_par_1000`	`stat_seq_1000`
ExaStamp-alone	75.66	75.66	75.66
Goldrush	81.90	92.73	131.53
SHC-1a-27s	77.35	86.05	86.01
SHC-7a-21s	99.00	99.67	101.20
DHC-7a-27s	78.17	81.76	85.84
DHC-27a-27s	79.00	82.29	85.85

Goldrush approach. For longer analytics, like the sequential statistics, the TINS approach with dynamic helper core strategy can be up to 34.74% faster than the Goldrush approach. The long execution time of the `stat_seq_1000` analytics reflects that Goldrush only manages to overlap with the simulation a small portion of the analytics because it executes analytics only during long enough sequential periods. The remaining of the analytics computations that Goldrush does not manage to execute during the simulation sequential sections is thus completed after the end of the simulation. The TINS task interleaving strategy prevents this issue by using both the simulation sequential periods and the periods when the simulation is not efficient enough to schedule analytics tasks.

Static versus Dynamic Helper Cores

In order to compare the different in situ strategies, we run a simulation of 256,000,000 particles with 64 MPI processes on 1,792 cores (Figs. 3 and 4). We tested various configurations to stress the memory accesses or the MPI communications for the `statistics` and the `histogram` routines. The `statistics` routines were executed 1 to 1,000 times at each analytics breakpoint. We present here only the results with 100 and 1,000 executions representative of the two main behaviors that emerged from these tests. The `histogram` was tested with global reductions applied on arrays of 1,000 to 1,000,000,000 integers. We include here the results for the intermediate size of 100,000,000 integers. For large arrays, execution times are similar for all strategies, dominated by the MPI communication. Analytics cost is too short with small array sizes to exhibit significant performance differences.

For each analytics, we tested various numbers of helper cores and arena sizes. `damaris-a-s` corresponds to Damaris running the analytics on `a` helper cores and the simulation on the remaining `s` cores. `SHC-a-s` (resp. `DHC-a-s`) corresponds to the TINS approach running the static (resp. dynamic) helper core strategy with an analytics arena of size `a` and a simulation arena of size `s`. Each histogram bar gives the total execution time of one strategy. A bar is divided into four areas: left part is the simulation master thread idle (no pattern) and active times (cross pattern); right part is the analytics master thread execution time split into idle (no pattern) and active times (dashed pattern).

For the sake of comparison, we implemented a synchronous version of the algorithm in Fig. 1 where the simulation master thread waits for the `analyticsDone` signal before computing the next iteration. We also implemented a pure asynchronous case where simulation and analytics tasks are created inside the same arena. Task scheduling is left to the TBB scheduler without any isolation or priority constraint. As a reference we also report the execution time of ExaStamp running on all cores without analytics, giving the best execution time we could expect if perfectly overlapping analytics with the simulation.

First, we notice that the TINS implementation presents a small overhead on the simulation execution time compared to the Damaris implementation. Our first studies tend to show that this overhead comes from the interaction between the two arenas and the observer in the TINS approach. Indeed, we interfere with TBB default data placement when using two arenas while there is only one arena in the Damaris case. This overhead depends on the number of helper cores but never exceeds 8%. On the other side, the execution of memory intensive analytics (`statistics_seq` and `statistics_par`) can be up to 75% longer with Damaris than with equivalent static helper core strategies implemented with TINS. Performance measurements with VTune show an important impact of NUMA effects, Damaris having up to 75% of DRAM remote accesses compared to 15% for TINS. Damaris relies on a shared memory segment managed by the Boost library and this shared memory segment is not bound to any specific memory bank. The shared memory segment can therefore be interleaved on different NUMA nodes, leading to performance penalties when data need to be accessed. TINS also relies on a copy but we do not need to create a shared memory segment because analytics and simulation belong to the same process. We tested various binding strategies for the temporary buffer. Compared to a situation where the buffer is not bound, the analytics is 41% quicker when binding the buffer on the NUMA nodes where the analytics worker threads belong and 22% longer when binding it on the NUMA nodes where the simulation worker threads belong. The different binding strategies do not have an impact on the simulation execution time and we decided to simply bind the buffer on the NUMA nodes where the analytics worker threads belong to speed up the analytics.

To compare the static helper core configurations, we can separate the analytics into two groups: the short analytics whose execution time are smaller than the simulation execution time (Fig. 3) and the long analytics whose execution time are equivalent or greater than the simulation execution time (Fig. 4). For short analytics, the best configuration is to dedicate one thread for the analytics. The analytics cannot benefit of any parallelism but the smallest number of threads are confiscated for the simulation and the analytics execution is still faster than the simulation iteration. Increasing the number of helper cores then leads to fewer threads for the simulation, which impacts the simulation execution time. For long analytics, the optimal number of static helper cores is analytics-dependent. Using 4 threads for `statistics_par` is a good trade off because if we use fewer threads, the analytics cannot benefit from its parallelization and

the total execution time is dominated by the analytics execution time. If we use more threads for the analytics, the simulation runs on fewer threads and the total execution time is dominated by the simulation execution time.

The dynamic helper core strategy is in general less sensitive to the configuration. For the small analytics in Fig. 3, there is less than 1% difference for the total execution time from one configuration to another. The different dynamic helper core configurations are therefore equivalent to a static helper core approach where one helper core is used. The analytics can be performed with an overhead of less than 5% with respect to ExaStamp alone and the dynamic helper core strategy can be up to 3% faster than the pure asynchronous approach and 28% faster than the synchronous approach that suffers from NUMA issues.

The results are similar with the sequential statistics performed 1,000 times (Fig. 4), with approximately 1% difference in the total execution time from one configuration to another. In the case of the parallel statistics performed 1,000 times (Fig. 4), setting an analytics arena of size 1 is too restrictive because the analytics cannot benefit from its parallelization. It therefore presents a total execution time 10% longer than the simulation alone while the other dynamic helper core configurations reduce this overhead to 6%. For these analytics, the dynamic helper core strategy is up to 40% better than the Damaris approach set with the appropriate number of static helper cores.

The radial analytics shows a slightly different behavior for the dynamic helper core strategy: increasing the concurrency level of the analytics arena also increases the total execution time. An analytics arena of size 1 induces an overhead of 6% with ExaStamp alone, this overhead growing up to 39% with an analytics arena of size 27. This analytics differs from the others because it executes two nested parallel loops. TBB does not support task switching on nested parallel loops. When a thread enters the analytics arena during simulation sequential periods, it cannot move back to the simulation arena before all the analytics tasks have been executed. In particular, it cannot switch back to support the simulation when the sequential region is over, slowing down the progress of the simulation. This effect is all the more visible as the analytics arena size increases. It is therefore necessary to reduce the size of the analytics arena in the dynamic helper core strategy, sizes of 4 and 7 being good tradeoff in this situation.

Experiments show that TINS implemented with the dynamic helper core strategy gives generally better performance than the static helper core strategy implemented by Damaris. In addition, our system shows greater flexibility for the choice of the number of helper cores, the execution times between the different dynamic configurations being relatively close.

Task versus Analytics Master Thread

TBB constrains master threads to execute only the tasks of the arena they created. Thus the analytics master thread never executes simulation tasks. As we spawn only $N - 2$ worker threads, there is always one core that cannot execute simulation tasks, potentially leading to underusing this core. We tried oversubscription by creating $N-1$ worker threads, but the performance degrades

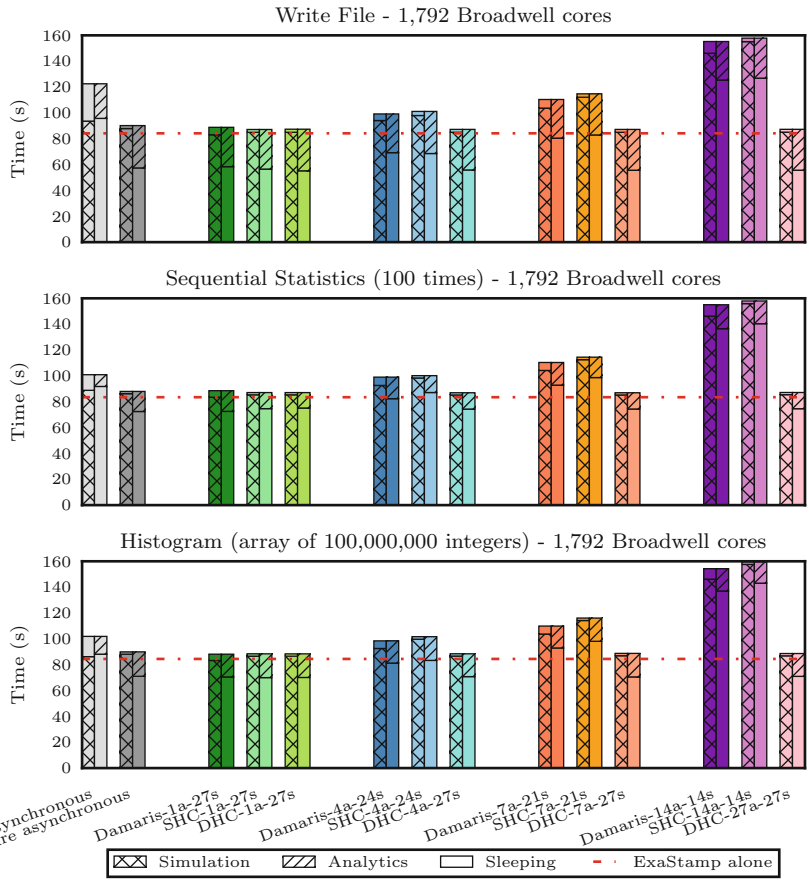

Fig. 3. Comparison of the different strategies on 1,792 Broadwell cores (64 MPI processes) for three analytics quicker than the simulation timestep: file writing (a), sequential statistics performed 100 times (b) and histogram with an array of 100,000,000 integers for the MPI collective communication (c).

significantly. To compare our analytics-master-thread approach with a version without additional master thread, we modified ExaStamp so that the simulation master thread creates an analytics task enqueued in the analytics arena task queue after data are copied. This task creates sub analytics tasks, as in the analytics-master-thread approach. The arena sizes are respectively set to N and n for simulation and analytics. The n threads in the analytics arena are pinned on the first cores, as in the static helper core strategy defined above.

Table 3 compares the execution times of the task approach (`task-a-s`) and the analytics master thread one (`thread-a-s`). The results are similar for the `histogram` computation (less than 2% of difference for the two methods) and the `radial` analytics (less than 4% of difference). However, the task method completely fails at reproducing the results of the thread method on the

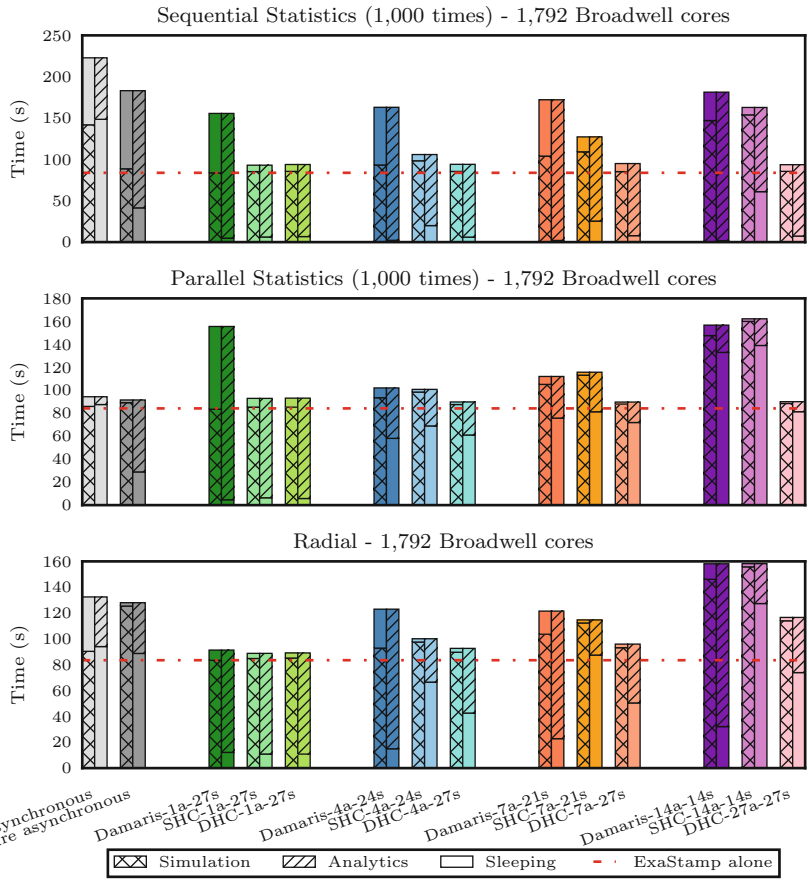

Fig. 4. Comparison of the different strategies on 1,792 Broadwell cores (64 MPI processes) for three analytics equivalent to or larger than the simulation timestep: sequential statistics performed 1,000 times (a), parallel statistics performed 1,000 times and radial.

`statistics_seq` analytics: the total execution time is up to 74% higher with an analytics arena of size 27. Performance measurements with VTune show that the percentage of DRAM remote accesses is of 18.5% with an analytics arena of size 7 and increases to 67.5% with an analytics arena size of 27 while it remains around 15% for TINS. In the thread approach, the sequential analytics will always be executed by the analytics master thread, guaranteeing data locality. In the task approach, we can bind the analytics threads on a set of cores but we cannot guarantee that the task will be executed on a particular thread. The task approach is also more intrusive in the simulation because the simulation needs to enqueue the task while it is left to a separate thread in the TINS approach. The TINS approach shows therefore better performance than a task approach and is less intrusive in the simulation.

Table 3. Execution times in seconds of the task and analytics-master-thread approaches, for three different analytics running the dynamic helper core strategy, with analytics arenas of sizes 7 and 27, on 1,792 Broadwell cores (64 MPI processes).

	`statistics_seq`			`radial`			`histogram`		
Time (s)	Total	Simulation	Analytics	Total	Simulation	Analytics	Total	Simulation	Analytics
task-7-27	110.7	102.6	94.6	93.6	91.6	29.9	90.7	89.0	23.7
thread-7-27	95.1	85.3	87.2	96.0	93.2	45.5	88.8	86.9	18.1
task-27-27	163.1	138.0	146.3	112.9	110.6	39.3	86.5	84.8	22.1
thread-27-27	93.7	85.6	86.3	116.6	114.0	42.6	88.6	86.9	17.6

Iteration Varying Analytics Workload

The analysis of simulation results often requires to execute different types of analytics at different iterations. Typically in production runs, different kinds of analytics are performed as the physics of the system evolves. To encompass this behavior, we execute 3 different statistics: the parallel statistics is computed during 10 iterations, the histogram is computed for the next 10 iterations and the radial distribution function is computed for the last 10 iterations.

Figure 5 compares the results for the different strategies on a simulation of 2 billions atoms using 14,336 Broadwell cores (512 MPI processes). The dynamic helper core approach always gives the best performance, being up to 20% faster than Damaris. As the analytics workload varies, no number of static helper cores is capable of ensuring the best performance for all the iterations. In opposite the dynamic helper core strategy offers more flexibility, leading to a better resource usage. Best results are obtained with an analytics arena of size 4 or 7 because the analytics can run in parallel and the simulation has still exclusive access to enough resources to ensure that its progression is not disturbed by analytics.

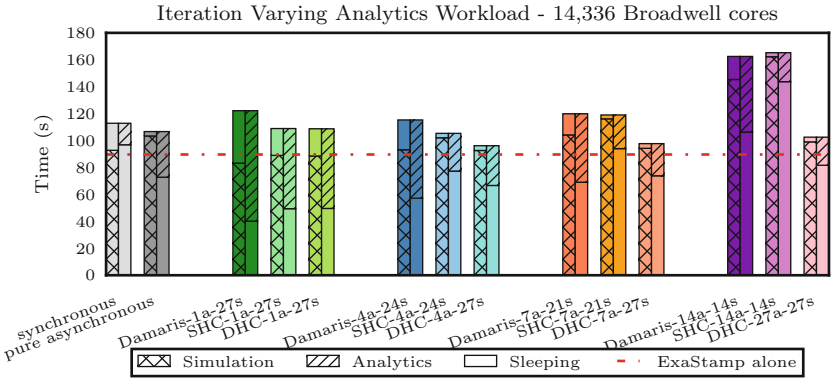

Fig. 5. Comparison of the different strategies on 14,336 Broadwell cores for an analytics scheme where the executed analytics depends on the iteration number.

5 Conclusion

Many previous works investigated how to perform asynchronous in situ processing at a process level for MPI applications. The helper core strategy emerged as the best approach to share the resources. In this paper, we propose the TINS approach that goes one step further by proposing a dynamic helper core strategy with a temporary thread isolation in a task-based programming model. The helper cores are assigned to analytics only when analytics tasks are available while they join the other threads for simulation processing instead. The TINS approach is a minimally intrusive method where it is easy to switch between static and dynamic helper core strategies without code recompilation and that is easy to use by the end-user. It enables use of both the simulation sequential regions and the part of the simulation that are not parallelized well enough. The experiments conducted on up to 14,336 Broadwell cores on representative analytics codes show that the TINS framework implemented with the Intel® TBB library can be up to 40% faster than the Damaris and Goldrush approaches on the ExaStamp molecular dynamics code that shows a good MPI and TBB efficiency. In particular, when the analytics workload varies from an iteration to another, no fixed number of static helper cores is capable of ensuring the best performance while the dynamic helper core strategy proves more flexible. Experiments also show that the obtained performance are close to the raw simulation, demonstrating that our approach enables to perform analytics at a high frequency. Future work will investigate the behavior of TINS on real analytics use cases. We also plan to study how to port TINS on other task-based runtimes, OpenMP in particular.

Acknowledgments. This work was partly funded by the French Programme d'Investissements d'Avenir (PIA) project SMICE. We thank Fang Zheng for having provided the Goldrush code and Matthieu Dorier for his help with Damaris.

References

1. Bennett, J.C., Abbasi, H., Bremer, P.-T., Grout, R., Gyulassy, A., Jin, T., Klasky, S., Kolla, H., Parashar, M., Pascucci, V., Pebay, P., Thompson, D., Yu, H., Zhang, F., Chen, J.: Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In: International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 49:1–49:9. IEEE Computer Society Press (2012)
2. Dreher, M., Raffin, B.: A flexible framework for asynchronous in situ and in transit analytics for scientific simulations. In: 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2014) (2014)
3. Dorier, M., Antoniu, G., Cappello, F., Snir, M., Orf, L.: Damaris: how to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O. In: IEEE International Conference on Cluster Computing (2012)
4. Zheng, F., Zou, H., Eisnhauer, G., Schwan, K., Wolf, M., Dayal, J., Nguyen, T.A., Cao, J., Abbasi, H., Klasky, S., Podhorszki, N., Yu, H.: FlexIO: I/O middleware for location-flexible scientific data analytics. In: IPDPS 2013 (2013)

5. Cieren, E., Colombet, L., Pitoiset, S., Namyst, R.: ExaStamp: a parallel framework for molecular dynamics on heterogeneous clusters. In: Lopes, L., et al. (eds.) Euro-Par 2014. LNCS, vol. 8806, pp. 121–132. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14313-2_11
6. Lorendeau, B., Fournier, Y., Ribes, A.: In situ visualization in fluid mechanics using Catalyst: a case study for Code_Saturne. In: IEEE Symposium on Large Data Analysis and Visualization (LDAV) (2013)
7. Fabian, N., Moreland, K., Thompson, D., Bauer, A., Marion, P., Geveci, B., Rasquin, M., Jansen, K.: The ParaView coprocessing library: a scalable, general purpose in situ visualization library. In: Large Data Analysis and Visualization Workshop (LDAV 2011), pp. 89–96 (2011)
8. Whitlock, B., Favre, J.M., Meredith, J.S.: Parallel in situ coupling of simulation with a fully featured visualization system. In: 11th Eurographics Conference on Parallel Graphics and Visualization, pp. 101–109 (2011)
9. Ayachit, U., Whitlock, B., Wolf, M., Loring, B., Geveci, B., Lonie, D., Bethel, E.: The SENSEI generic in situ interface. In: 2nd Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV 2016), pp. 40–44 (2016)
10. Lofstead, J.F., Klasky, S., Schwan, K., Podhorszki, N., Jin, C.: Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In: 6th International Workshop on Challenges of Large Applications in Distributed Environments, pp. 15–24 (2008)
11. Zheng, F., Yu, H., Hantas, C., Wolf, M., Eisenhauer, G., Schwan, K., Abbasi, H., Klasky, S.: GoldRush: resource efficient in situ scientific data analytics using fine-grained interference aware execution. In: International Conference on High Performance Computing, Networking, Storage and Analysis (SC 2013), pp. 78:1–78:12 (2013)
12. Harris, T., Maas, M., Marathe, V.J.: Callisto: co-scheduling parallel runtime systems. In: Proceedings of the Ninth European Conference on Computer Systems (EuroSys 2014), pp. 24:1–24:14 (2014)
13. Li, M., Vazhkudai, S.S., Butt, A.R., Meng, F., Ma, X., Kim, Y., Engelmann, C., Shipman, G.: Functional partitioning to optimize end-to-end performance on many-core architectures. In: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12 (2010)
14. Singh, A., Balaji, P., Feng, W.: GePSeA: a general-purpose software acceleration framework for lightweight task offloading. In: International Conference on Parallel Processing, pp. 261–268 (2009)
15. Ma, X., Lee, J., Winslett, M.: High-level buffering for hiding periodic output cost in scientific simulations. IEEE Trans. Parallel Distrib. Syst. **17**(3), 193–204 (2006)
16. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput. Pract. Exper. **23**, 187–198 (2011)
17. Hoque, R., Herault, T., Bosilca, G., Dongarra, J.: Dynamic task discovery in PaRSEC: a data-flow task-based runtime. In: Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA 2017, pp. 6:1–6:8. ACM, New York (2017). http://doi.acm.org/10.1145/3148226.3148233
18. 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 (SC 2012) (2012)

19. 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 (PGAS 2014) (2014)
20. Pébaÿ, P., Bennett, J.C., Hollman, D., Treichler, S., McCormick, P.S., Sweeney, C.M., Kolla, H., Aiken, A.: Towards asynchronous many-task in situ data analysis using legion. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1033–1037, May 2016
21. Heirich, A., Slaughter, E., Papadakis, M., Lee, W., Biedert, T., Aiken, A.: In situ visualization with task-based parallelism. In: Workshop on In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization (ISAV 2017) (2017)
22. Cho, Y., Oh, S., Egger, B.: Adaptive space-shared scheduling for shared-memory parallel programs. In: Desai, N., Cirne, W. (eds.) JSSPP 2015-2016. LNCS, vol. 10353, pp. 158–177. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61756-5_9
23. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999)
24. Dorier, M., Sisneros, R., Peterka, T., Antoniu, G., Semeraro, D.: Damaris/Viz: a nonintrusive, adaptable and user-friendly in situ visualization framework. In: IEEE Symposium on Large Data Analysis and Visualization (LDAV) (2013)