GoldRush: Resource Efficient In Situ Scientific Data Analytics Using Fine-Grained Interference Aware Execution

Fang Zheng[1], Hongfeng Yu[2], Can Hantas[1], Matthew Wolf[1,3],
Greg Eisenhauer[1], Karsten Schwan[1], Hasan Abbasi[3], Scott Klasky[3]
[1]Georgia Institute of Technology [2]University of Nebraska Lincoln [3]Oak Ridge National Laboratory

ABSTRACT

Severe I/O bottlenecks on High End Computing platforms call for running data analytics in situ. Demonstrating that there exist considerable resources in compute nodes un-used by typical high end scientific simulations, we leverage this fact by creating an agile runtime, termed GoldRush, that can harvest those otherwise wasted, idle resources to efficiently run in situ data analytics. GoldRush uses fine-grained scheduling to “steal” idle resources, in ways that minimize interference between the simulation and in situ analytics. This involves recognizing the potential causes of on-node resource contention and then using scheduling methods that prevent them. Experiments with representative science applications at large scales show that resources harvested on compute nodes can be leveraged to perform useful analytics, significantly improving resource efficiency, reducing data movement costs incurred by alternate solutions, and posing negligible impact on scientific simulations.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Scheduling.

General Terms

Design, Measurement, Performance.

1. INTRODUCTION

Many large scale scientific simulations can routinely write out immense amounts of data on today’s High End Computing platforms. Such “Big Data” imposes steadily increasing pressure on the I/O and storage sub-systems. In fact, I/O is now widely recognized as a severe performance bottleneck for both simulation and data post-processing; and this is expected to worsen with expected order of magnitude increases in the disparity between computation and I/O capacity on future Exascale machines [35].

In order to mitigate the I/O bottleneck, leadership scientific applications (e.g., GTS [41], S3D [9], and FLASH [39]) have begun to use in situ data analytics, where analytics are deployed on the same HEC platform where the simulation runs, with simulation output data processed online while it is being generated. Compared to conventional post-processing methods that first write data to storage and then read it back for analysis, in situ analytics can reduce on-machine data movement, disk I/O volume, and deliver faster insights from raw data [2].

The research presented in this paper has two goals: (1) to improve the resource efficiency of running in situ data analytics, and (2) to do so without perturbing the simulations running on the same nodes. In particular, we seek to over-subscribe compute nodes by co-locating simulation and analytics computations, without affecting the simulation execution, while at the same time, efficiently using compute node resources to run in situ analytics.

Measurements of six representative scientific simulations motivate the argument that node over-subscription can be cost neutral to the core simulation. Specifically, we demonstrate that the well-tuned MPI/OpenMP implementations of these codes written for high end machines leave substantial unused resources (CPU and memory) on compute nodes, which can then be used to run online analytics. One cause is sequential periods in these codes (i.e., when the execution flow is outside their OpenMP parallel regions) in which worker threads wait on the MPI process’ main thread. Although most such sequential periods are short, their aggregate duration can be up to 65% of total execution time in these real-world codes.

Previous work has sought to reduce sequential periods and utilize spare node resources by overlapping the main thread’s sequential work with OpenMP regions, but such application-specific tuning efforts are limited by data and control dependencies, and they can also impede code clarity and portability. In fact, none of the six codes in our study uses such overlapping in their production versions. The novel “GoldRush” method presented in this paper uses a different approach to exploiting idle node resources: it uses them to run the in situ data analytics needed to cope with I/O bottlenecks. Benefits include the efficient use of compute node resources and reductions in data movement overheads, as will be demonstrated with detailed performance measurements.

The GoldRush method is made possible by the FlexIO transport in the ADIOS I/O system [19][47] widely used on high end machines. Specifically, with FlexIO and ADIOS, analytics pipelines can be configured to map to compute nodes only those portions of their computations that “fit into” available idle resources, with additional analytics mapped to dedicated resources and/or run as post-processing tasks after data has been moved to the machine’s attached parallel file system. Appropriate end-to-end mappings of analytics pipelines can reduce I/O data volumes and data movement overheads [1][45][47], to provide science end users with rapid insights into the data produced by their simulations.

SC13 November 17-21, 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11...$15.00.
http://dx.doi.org/10.1145/2503210.2503279

Leveraging such flexibility in constructing data analytics pipelines, this paper addresses the key compute-node-level challenges for efficiently running in situ data analytics. The first challenge is that for well-tuned scientific simulations, idle compute-node CPU cycles exist in the form of a large number of short idle periods. This makes it difficult to schedule and allocate cores to analytics without causing undue runtime overheads for the simulation. Second, because co-located simulation and analytics codes share certain node resources (e.g., last level caches, memory busses and controllers), the execution of analytics must be managed to minimize the degrees to which simulations are perturbed. Measurements presented in this paper demonstrate that carefully managing how analytics are run is critical to achieving overall high performance for co-located simulation and analytics. Third, current operating systems on HEC platforms are not well equipped to deal with multi-programmed simulation and analytics workloads, as they schedule processes based on core idleness, essentially allocating idle resources to analytics in a greedy manner, and they are also largely ignorant of potential interference effects. Therefore, even with carefully configured process priorities, such policies can lead to severe performance loss. As shown later, priority-based OS level scheduling of analytics processes can result in an up to 57% performance degradation of the simulations.

To address those challenges, we have created a lightweight runtime system, named "GoldRush", which supports resource-efficient and non-intrusive in situ data analytics. GoldRush (i) uses low-overhead online monitoring to identify opportunity windows during which (ii) it can schedule analytics to run on cores not currently used by the simulation. It also (iii) continuously assesses interference between concurrently-running simulation and analytics, and (iv) controls the execution rate of analytics processes to mitigate harmful impacts on the simulation due to contention on shared node resources.

GoldRush makes the following contributions:

1) Fine-Granularity Operation: during simulation execution, it identifies idle periods, predicts the duration of each period, selects those periods with sufficient durations to run analytics, but skips those that are too small to dwarf context switching overheads. It completely suspends analytics when cores are in use by the simulation, to avoid perturbing the parallel simulation.

2) Interference Awareness: it can detect interference between concurrently running simulation and analytics arising from contention on shared memory resources, and it dynamically mitigates such interference by throttling the execution rate of analytics.

3) Low Overhead: runtime overheads (including monitoring and scheduling) are negligible, measured as never exceeding 0.3% of total runtime with representative HEC applications.

4) Transparency: its methods are easily integrated into existing HEC runtimes, demonstrated by their use with OpenMP/MPI hybrid codes, thus imposing minimal restrictions on current simulation and analytics codes.

By effectively managing co-located simulation and analytics workloads, GoldRush complements existing in situ data analytics techniques [2][6][7][42][46], opening up new opportunities to efficiently run such analytics without the need to dedicate compute node resources, leading to substantial performance improvements and cost savings at large scales.

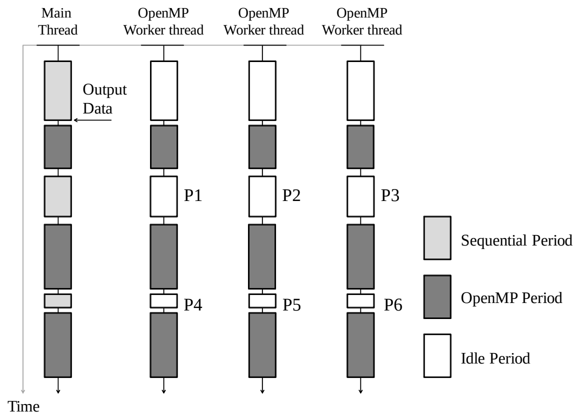


Figure 1. Illustration of idle resources during execution of a MPI processes with 4 OpenMP threads. The 3 OpenMP worker threads are idle when the main thread is in sequential periods.

GoldRush is evaluated with real-world scientific applications on NERSC's Hopper Cray XE6 and Oak Ridge National Laboratory' InfiniBand cluster. In particular, measurements with co-located simulation and synthetic analytics show that GoldRush's synergistic scheduling improves simulation performance by 9.9% on average (and up to 42%) over the OS scheduling. For a fusion application GTS, there is a clear trend that GoldRush's advantage over the OS baseline native scheduling methods increases at larger scales (up to 7.5% at 12288 cores); and that the GoldRush-managed analytics outperforms alternative analytics setups: for GTS at 12K cores, it achieves 30% performance improvement over "Inline" analytics and a 1.8x reduction in data movement volumes over "In-Transit" analytics. Additional evaluations on a 32-core, multi-socket Intel Westmere machine demonstrate GoldRush's node-level scalability and applicability across different architectures.

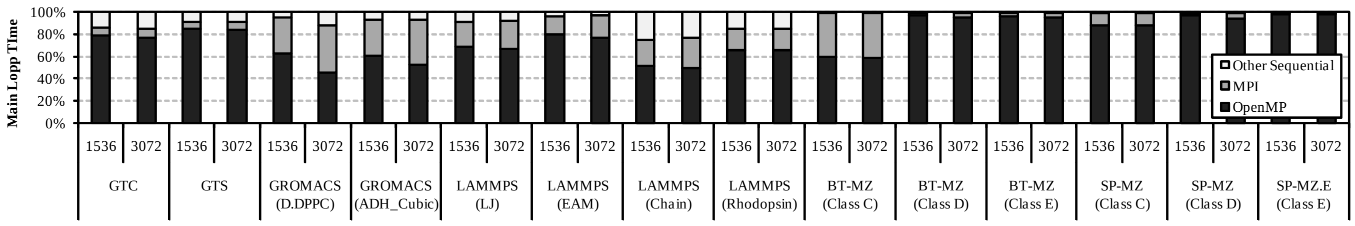
The remainder of the paper is organized as follows. Section 2 motivates GoldRush with experimental measurements that show the benefits and challenges of leveraging idle compute node resources for in situ data analytics. Section 3 describes the system design and implementation of GoldRush and the techniques used to gain high levels of performance and resource efficiency. Section 4 evaluates GoldRush with both synthetic benchmarks and real-world applications on different HEC platforms. Section 5 reviews related work and Section 6 concludes the paper.

2. MOTIVATION

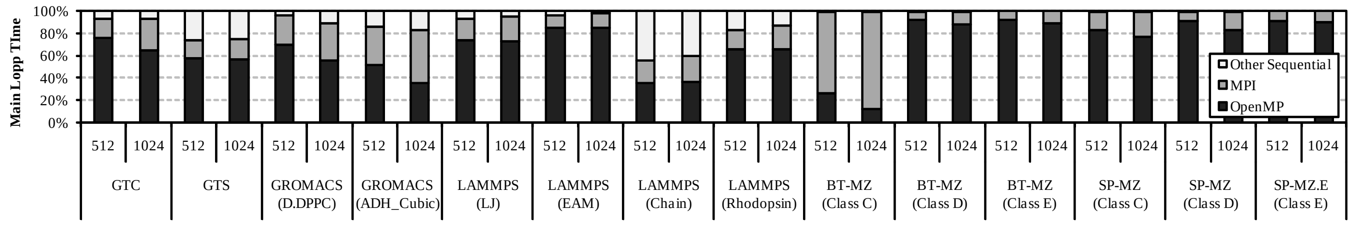
This section presents a detailed characterization of the idle resources on compute nodes, to quantify the potential benefits and challenges of using them.

2.1 Characterizing Idle Resources

Figure 1 illustrates the execution of a MPI process with multiple OpenMP threads. When only the main thread in the MPI process is actively executing some sequential code outside OpenMP regions (i.e., in sequential periods), the OpenMP worker threads are waiting and the cores on which they run become idle ("P1" to "P6" in Figure 1). Typical sequential periods involve MPI communications, file I/O, and/or non-parallelized computations. Analytics can be run asynchronously, in response to a simulation's data output action and using available idle cores, as long as there is sufficient free memory for buffering output data between successive simulation output actions.



(a) On Hopper, simulations run on 1056 (256 MPI proc. × 6 OpenMP threads) and 3072 cores (512 MPI proc. × 6 OpenMP threads).



(b) On Smoky, simulations run on 512 (128 MPI proc. × 4 OpenMP threads) and 1024 cores (256 MPI proc. × 4 OpenMP threads).

Figure 2. Breakdown of simulation main loop time. The input decks are specified in parentheses following the simulation names. When the simulation is in non-threaded sequential periods, only its main thread is active and OpenMP worker threads are idle.

We are interested in how many idle resources (CPU and memory) exist when running real-world HEC simulation codes and whether those idle resources are amenable for use by in situ analytics. Toward that end, we profile four widely-used and well-tuned MPI/OpenMP hybrid simulation codes: GTC (fusion) [13], GTS (fusion) [41], GROMACS (molecular dynamics) [8], LAMMPS (molecular dynamics) [28], plus two well-known MPI/OpenMP hybrid benchmark codes: BT-MZ and SP-MZ from the NPB benchmark suite [22].

The six codes are profiled on NERSC's Hopper Cray XE6 [10] and on ORNL's Smoky InifiniBand cluster [34]. Hopper has 6,384 compute nodes and uses Cray's Gemini interconnect. Each Hopper compute node has two 12-core MagnyCours AMD processors. There are 4 NUMA domains, each with 6 cores and 8GB DRAM. Smoky is an 80 node cluster, where each compute node has four quad-core AMD Opteron processors. There are 4 NUMA domains, and each domain has 4 cores and 8GB DRAM. To accommodate the NUMA architecture, we run each MPI process in one NUMA domain and run as many OpenMP threads as the number of cores in each NUMA domain (which leads to peak performance for all simulation codes). Threads are pinned on cores, and memory affinity is enforced within each NUMA domain with the aprun and mpirun launch facility.

GTC, GTS, BT-MZ and SP-MZ are built with the PGI compiler, and GROMACS and LAMMPS with the GCC compiler, respectively (as suggested by the developers). Codes are run with representative input configurations, and GROMACS, LAMMPS, BT-MZ, and SP-MZ are run with the multiple input decks distributed with these software packages. The CrayPAT [4] and Vampir [38] tools are used to collect profiling information.

Each simulation's main loop time is divided into three parts: (1) OpenMP periods (all threads are active), (2) MPI periods (only the main thread is active, performing MPI communications), and (3) "Other Sequential" periods (only the main thread is active, carrying out sequential activities like file I/O or others). In the latter two cases, the cores on which OpenMP worker threads run are idle. Figure 2 shows the percentages of execution time spent in those three parts.

Interesting observations from these measurements include the following. First, jointly, all idle periods (MPI and Other Sequential periods) comprise up to 65% of the total main loop time for four of these applications (i.e., LAMMPS with the "Chain" input deck), and even 89% for the NPB BT-MZ benchmark with the class C input. Note that on Hopper's compute nodes, 20 out of 24 cores are idle during those periods, leading to substantial amounts of idle compute capacities. Second, the percentage of total idle periods generally increases when scaling the simulation to run on more cores. For example, GTC's idle period percentage increases from 21% to 23% when scaling from 1536 to 3072 cores on Hopper. This holds for weak scaling codes like GTC, GTS, and LAMMPS in which MPI communication times increase at larger scale, and also for strong scaling codes like GROMACS and the NPB benchmarks, where in OpenMP times decrease with increased core counts. Third, although simulation performance varies across inputs (like LAMMPS and GROMACS), it is common that idle periods comprise a substantial portion of total simulation runtime.

We also measure peak memory usage among all MPI processes. None of the simulation codes consume more than 55% on either Hopper or Smoky. The resulting available free memory makes it feasible to buffer simulation output data, thereby enabling the asynchronous execution of analytics and simulation codes.

2.2 Challenges of Using Idle Resources

Although the measurements shown so far demonstrate sufficient availability of idle resources, there are several challenges for effectively harvesting these idle resources for in situ data analytics, discussed next.

2.2.1 Magnitude of Idle Resources

Despite the substantial amounts of total idle CPU cycles, most individual idle periods are short in duration. Figure 3 shows the distribution of durations of idle periods in our six codes. The "Count" histograms show that for all simulation codes, the majority of idle periods are quite short (less than 1ms), while the "Aggregated Time" histograms show that the total amount of idle time is dominated by a modest number of large idle periods.

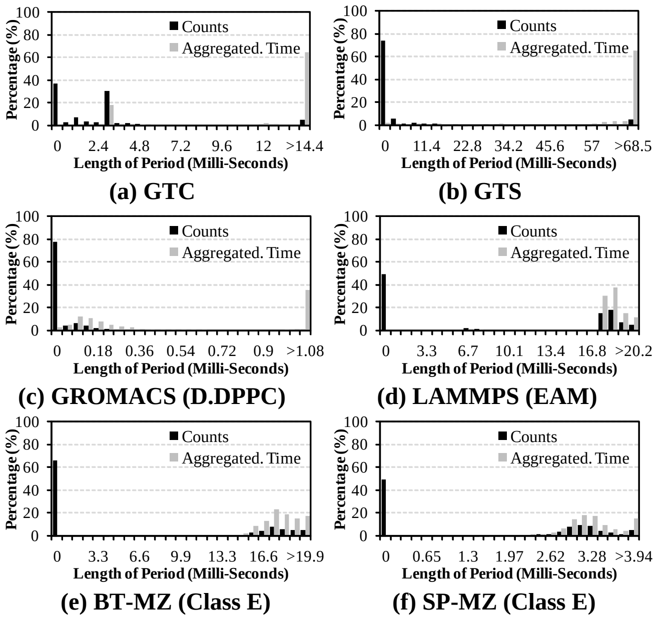


Figure 3. Distribution of idle period duration. All simulations run with 1536 cores on Hopper.

This distribution pattern has important implications. First, it is not likely useful, in terms of cost vs. benefit, to harvest small idle periods. As a result, one must determine, at runtime, which idle periods will be sufficiently large to warrant their use for running desired data analytics. Second, inaccurate methods for identifying appropriately long idle periods will lead to inefficiencies for two reasons: (1) insufficient benefits or worse, undue overheads when using periods that are too small, and (2) missed larger periods leading to loss of major portions of total available idle time.

2.2.2 Contention on Shared Resources

Beyond finding idle periods suitable for running analytics, another issue is the potential interference of analytics imposed on the simulation's main thread running in its sequential phase (during which analytics processes concurrently run on idle cores not used by the simulation's OpenMP worker threads). Interference is due to contention on resources shared between both sets of threads, such as the last level cache, the memory bus, and the memory controller (as shown in Figure 4); it is particularly harmful for tightly synchronized parallel simulations, as the slowdown of each individual MPI process may cascade and be amplified when running at larger scales [11].

2.2.3 Limitations of Operating System Scheduling

A baseline solution for co-running analytics with simulation threads is to leave it to the Linux OS scheduler and the OpenMP runtime to manage both workloads. We realize this approach as follows.

1) On each compute node, fork some number of analytics processes. Set their CPU affinities so that they can run on the cores where the simulation's OpenMP worker threads are run, but not on the cores hosting the simulation's main threads. The analytics processes are given the lowest priority (with "nice" values set to 19).

2) Configure the simulation's OpenMP runtime so that worker threads yield CPUs when they are outside OpenMP regions. For the Intel OpenMP runtime, this can be achieved by setting the KMP_BLOCKTIME environment variable to 0. The PGI and GNU OpenMP runtimes can be similarly configured, by setting the OMP_WAIT_POLICY environment variable to "PASSIVE". The priorities of the simulation's OpenMP worker threads are set to default (their "nice" values are equal to 0).

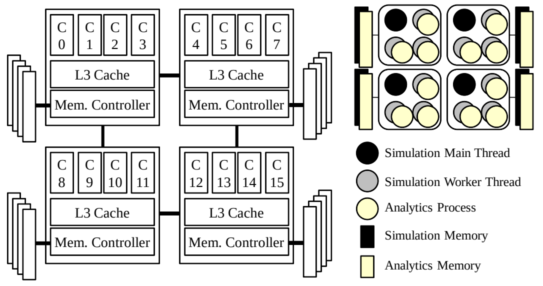
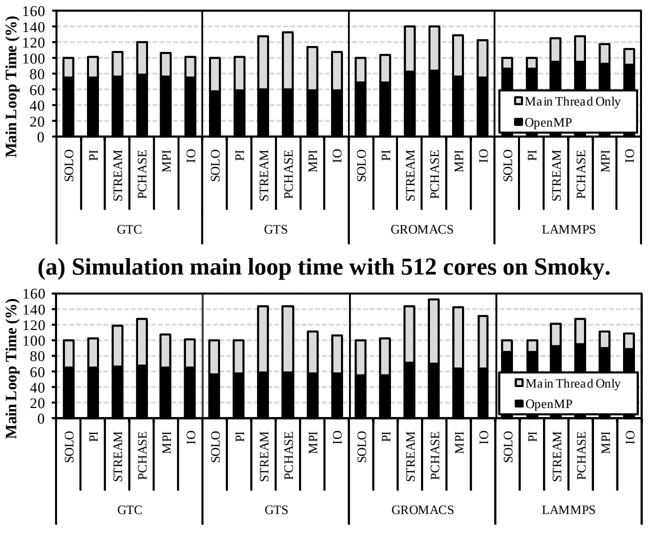


Figure 4. Placement of simulation and in situ data analytics on Smoky's 16-core compute node.

Table 1. Analytics Benchmarks

Benchmark	Tasks for Each Process
PI	Iteratively calculate Pi.
PCHASE	Traverse randomly linked lists (200MB in total).
STREAM	Sequentially scan large arrays (200MB in total).
MPI	Collectively call MPI_Allreduce() on 10MB data.
IO	Write 100MB data to parallel file system.



(a) Simulation main loop time with 512 cores on Smoky.

(b) Simulation main loop time with 1024 cores on Smoky.

Figure 5. Simulation performance with co-located analytics.

This baseline solution is evaluated by co-running the six simulations with the five analytics benchmarks listed in Table 1. These benchmarks each stress a certain subsystem in the machine. On Smoky, we run each simulation with 512 cores (128 MPI processes and 4 OpenMP threads per process) and with 1024 cores (256 MPI processes, each with 4 OpenMP threads). In both cases, there are 16 simulation threads and 12 analytics processes on each compute node, as shown in Figure 4.

Figure 5 shows the performance of four simulations with co-running analytics. Each simulation's main loop time is divided into two parts: parallel OpenMP periods and Main-Thread-Only periods (the latter correspond to MPI and Other Sequential periods in Figure 2). With the pure OS-based management solution, co-located analytics slow down simulations by up to 57% compared to simulations' solo runs, and performance degradation generally becomes worse at larger scales.

The ineffectiveness of pure OS-based management is caused by several factors. First, the significant slowdown of the Main-Thread-Only periods shown in Figure 5 indicates that the simulation's main threads experience severe interference from concurrently running analytics. This is particularly true for cases in which the simulation's main threads co-run with memory intensive codes like PCHASE and STREAM, because those benchmarks cause severe contention on the last level cache, memory controller, and other shared resources in the memory hierarchy. Linux' default OS scheduler does not recognize those facts, as its main focus is on core idleness.

Second, there are increases in some simulations' OpenMP times with the presence of co-located analytics. One reason is the OS scheduler's greedy nature, which always schedules analytics threads as soon as the OpenMP worker threads yield the CPU. For short idle periods, analytics threads will be forced to suspend soon after they begin to run, to return cores back to higher priority simulation threads. Another reason is the Linux scheduler's imposition of fairness on analytics vs. simulation threads, causing it to allocate time slots for, rather than completely suspend, low-priority analytics processes while the simulation's worker threads are active (i.e., in a parallel OpenMP period). This causes jitter to the simulation and negatively impacts its performance.

The GoldRush runtime methods described next remedy these shortcomings of the OS baseline solution.

3. GOLDRUSH RUNTIME SYSTEM

3.1 Overview

GoldRush manages the execution of data analytics co-located with simulation processes, in ways that (i) leverage unused idle resources on compute nodes, and (ii) mitigate potential interference between simulation and analytics.

GoldRush is implemented as a runtime library and residing at both the simulation and analytics sides of these compute node-based computations (highlighted in yellow in Figure 6). For simulation processes, GoldRush generates performance monitoring metrics used by a prediction module to estimate the lengths of upcoming idle periods at the exit of each OpenMP parallel region. If the next idle period is predicted to be "usable", GoldRush sends signals to analytics processes to resume their execution; if no signal is produced, analytics processes remain suspended throughout the next idle period. Once resumed, analytics processes run on the cores yielded by the simulation's OpenMP worker threads, while the simulation's main threads continue to run on their own, dedicated cores. When the simulation's main threads reach the end of their idle periods (i.e., the start of next parallel OpenMP region), signals are sent to suspend analytics processes, thereby permitting the simulation's OpenMP worker threads to re-gain exclusive use of their cores for executing the subsequent parallel OpenMP period.

To assess potential interference between simulation and analytics, the GoldRush runtime also periodically updates a shared memory monitoring buffer with performance data about the simulation's main threads. The analytics-side GoldRush scheduler periodically

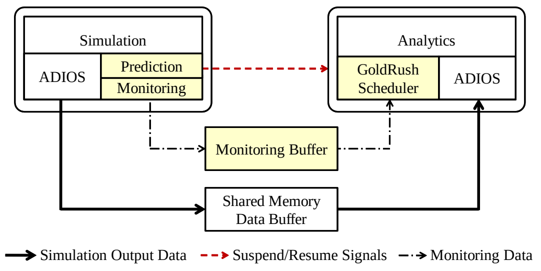


Figure 6. Architecture of GoldRush runtime.

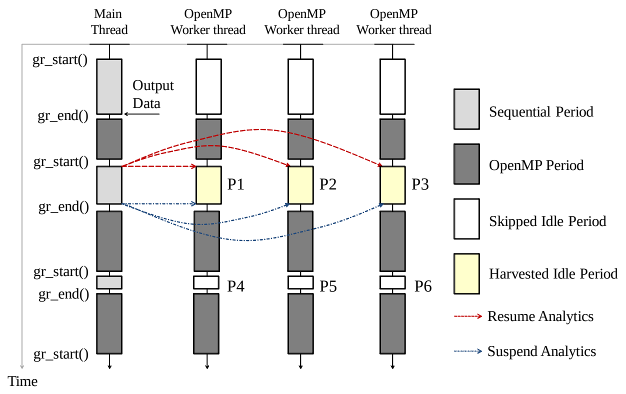


Figure 7. Simulation and analytics execution timeline.

reads this information, assesses interference severity and if significant interference is detected, the scheduler throttles, i.e., slows down, the execution rate of analytics processes. This serves to reduce contention on shared resources, at the cost of reduced progress with analytics processing. A limit on possible slowdown is imposed by the fact that analytics processing must be completed before the simulation's next output steps are taken. On-compute-node analytics, therefore, have to be "sized" appropriately, and we do so by leveraging the placement flexibility offered by the ADIOS IO library and its FlexIO IO methods, described in [47]. With ADIOS and FlexIO, analytics pipelines can be defined and (re-)structured to match available compute node resources, with "overflow" analytics actions performed in separate "staging nodes" reserved for online analytics and/or postmortem, after data has been moved to disk. Another attribute of the FlexIO transport used by GoldRush is its efficient intra-node data movement from simulation to analytics via a shared memory transport.

Compared to the baseline solution described in Section 2, GoldRush adds potential overheads to the simulation side for performance monitoring and idle period prediction, and for suspending and resuming analytics. There are also additional costs at the analytics side for online monitoring and execution control. As shown in Section 4, these overheads are negligible, permitting GoldRush to significantly improve application performance and resource efficiency over the baseline solution.

3.2 Inter-Posing GoldRush

GoldRush is implemented as a C library, for which we offer two approaches to integrating it with simulation codes. The first approach directly inserts the GoldRush API (listed in Table 2) into the simulation's source code. In particular, a gr_start() call is placed at the end of an OpenMP code region (e.g., after a "!$omp end parallel" statement) to mark the start of an idle period; and a gr_end() call is put before the beginning of an OpenMP parallel

Table 2. GoldRush Public API

Function	Description
int gr_init (MPI_Comm comm);	Initialize the GoldRush runtime
int gr_start (char *file, int line);	Mark the start of an idle period
int gr_end (char *file, int line);	Mark the end of an idle period
int gr_finalize ();	Finalize the GoldRush runtime

region (e.g., before a "!$omp parallel" statement) to mark the end of an idle period. At runtime, those markers are executed by the main thread of each simulation process to identify the beginning and end of idle periods, and to perform operations that monitor performance and resume/suspend analytics processes.

The second approach integrates the library with the simulation in a more transparent fashion, avoiding changes to simulation codes, by adding its functions into appropriate routines within the OpenMP runtime library. As a proof of concept, we modify GCC's libgomp runtime library by instrumenting the runtime routines associated with PARALLEL and FOR directives. Those are sufficient to cover all of the top-level OpenMP regions in the GTC, GTS, LAMMPS, GROMACS, and NPB codes. Other directives can be supported similarly, left for future work.

In comparison, the source code instrumentation approach is more general and flexible at the cost of manual code modification. The instrumented OpenMP runtime library approach is transparent to simulation codes, but requires modifying internals of the OpenMP library. In practice, we have instrumented the sources of simulation codes requiring Intel or PGI compilers, as those compilers' OpenMP runtime libraries are not available to us for modification. Besides, source instrumentation may be automated with source transformation [23] or binary re-writing tools [22].

Analytics codes only need to add gr_init() and gr_finalize() functions, permitting an instance of the GoldRush scheduler to be activated in each analytics process at runtime.

3.3 Online Monitoring and Prediction

3.3.1 Predicting Idle Period Durations

At the beginning of an idle period (i.e., in a gr_start() call), the simulation's OpenMP worker threads have yielded their cores, and the main thread is about to enter a sequential code region. An important decision to make at this point is: *should the analytics processes be allowed to run on idle cores during this upcoming idle period?* As discussed in Section 2.2, idle periods are appropriate only if they are sufficiently long. To predict their expected durations, the GoldRush runtime records the timings and number of occurrence of each executed idle period. Each idle period is uniquely identified by its start and end locations (the file name and line number arguments passed to marker API calls). When a gr_end() marker is executed, the idle period that just completed is identified. The duration of that idle period is measured as the elapsed time between the two successive gr_start() and gr_end() calls made by the main thread. The online history maintains a running average duration and occurrence counts for each unique idle period seen so far.

We currently use a simple heuristic to predict idle period duration, using the above online history information. The method has high accuracy and low overheads for simulations with strong locality and regularity in their execution flows (a typical behavior for many scientific codes), as those codes usually have a small number of unique idle periods with small variations in idle period duration. The heuristic works as follows. During the execution of gr_start(), a prediction function is called. It first finds all idle periods in the history that match the start location (file name and line number) of the upcoming idle period, selects the one with the highest occurrence count, and uses its running average duration as an estimate of the upcoming period's duration. If the estimated duration is greater than a pre-defined, tunable threshold value or no matching history record is found, the upcoming idle period is considered as "usable" for analytics.

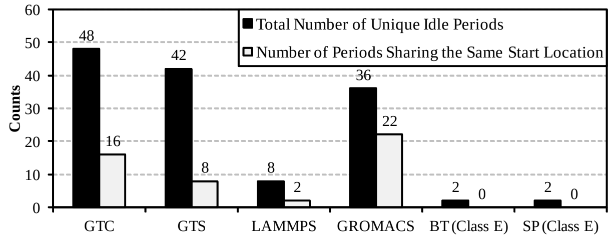


Figure 8. Number of unique idle periods and idle periods with the same start location (due to branching in execution flow).

Table 3. Prediction Accuracy with 1ms Threshold (1536 Cores on Hopper).

Simulation	Predict Short	Predict Long	Mispredict Short	Mispredict Long
GTC	31.6%	57.1%	6.4%	4.9%
GTS	58.5%	36.8%	3.6%	1.1%
LAMMPS	49.7%	49.7%	0.3%	0.3%
GROMACS	99.6%	0.1%	0.1%	0.2%
BT-MZ.E	66.6%	33.4%	0.0%	0.0%
SP-MZ.E	50.1%	49.9%	0.0%	0.0%

Costs: The time and space costs of idle period prediction are proportional to the number of unique idle periods in a simulation's execution flow. As shown in Figure 8, the numbers of unique idle periods in the six simulation codes range from 2 to at most 48, resulting in low runtime overheads.

Prediction Accuracy: The purpose of prediction is to decide whether an idle period is usable (long) or not (short) with respect to a threshold value. Therefore, instead of using the absolute error in predicted duration values, we define a prediction of an idle period to be "accurate" if the predicted usability (short or long) of the idle period matches the indication of the actual duration. Specifically, we divide prediction results into four categories: (i) "Predict Short": correctly predict a short period to be short (not usable for analytics); (ii) "Predict Long": correctly indicate a long period to be long (usable); (iii) "Mispredict Short": wrongly predict a short period to be long; and (iv) "Mispredict Long": wrongly predict a long period to be short.

To quantify prediction accuracy, we record the predicted duration at the beginning of each idle period, and measure the actual duration at the end of the period, based on which we then count the number of predictions falling into each of the four categories described above. Table 3 presents the percentages of the four categories among all predicted periods, using a threshold value of 1ms. Accurate predictions range from 88.7%~100% of all predictions for the six simulations, showing that our prediction method is highly accurate for codes with regular execution flows.

Figure 9 shows how sensitive prediction accuracy is to the threshold value. When varying the threshold value from 0.1 to 2 milliseconds, prediction accuracy for all six simulations never falls below 84.5%, and remains 100% for BT-MZ and SP-MZ cases. Figure 9 also shows that 1ms is an appropriate threshold value since it leads to high accuracy and in addition, ensures that the selected usable periods are sufficiently large to amortize context switch overheads.

Despite good results with the six simulation codes used in our work, there remain substantial opportunities for future improvements and optimizations of methods for idle period prediction. For instance, for codes with dramatically varying idle periods and runtimes (e.g., Adaptive Mesh Refinement codes), more sophisticated methods like dynamic call stack tracking plus statistical forecasting are likely preferable, which we will investigate as future work.

3.3.2 Monitoring Interference during Idle Periods

To manage potential interference between a simulation's main threads and concurrent analytics processes, GoldRush installs a timer and signal handler on each main thread to inspect relevant hardware performance counter values through the PAPI performance counter library [26], done every millisecond during idle periods. Measured are the number of CPU cycles and retired instructions, and IPC (Instructions per Cycle) is calculated to quantify the performance of the simulation's main thread. The IPC value is written to a per-simulation-process buffer in shared memory, and is periodically read by the analytics-side GoldRush schedulers. The timer is disabled at the end of each idle period.

3.4 Controlling Execution of Analytics

Analytics are run when an idle period is predicted as usable. This involves the simulation main thread sending a SIGCONT signal to resume the execution of analysis processes. Conversely, when the simulation main thread calls gr_end() at the end of the idle period, it sends a SIGSTOP signal to suspend analytics. Analytics threads, therefore, are run only during selected idle periods; they are quiescent when the simulation is in its OpenMP regions. The signaling costs incurred are small (see Section 4).

An alternative to using signals to suspend and resume analytics processes is to set the simulation processes to use a real-time scheduling policy via the sched_setscheduler() system call. However, this privileged feature is not generally available in HPC environments (e.g., Hopper and ORNL's Titan Cray XK7).

3.5 Scheduling Analytics

At the analytics side, the GoldRush scheduler regulates the execution of analytics processes to mitigate potential interference effects experienced by the simulation's main threads. The scheduler is implemented as a signal handler in each analytics process and is periodically triggered by a timer. Two scheduling policies are presented below.

3.5.1 Interference-Aware Policy

The Interference-Aware scheduler works in three steps.

1) Assessing the Severity of Interference: once triggered, the scheduler reads the IPC value of the simulation's main thread from the shared memory monitoring buffer. Interference is determined as IPC being lower than some threshold value,

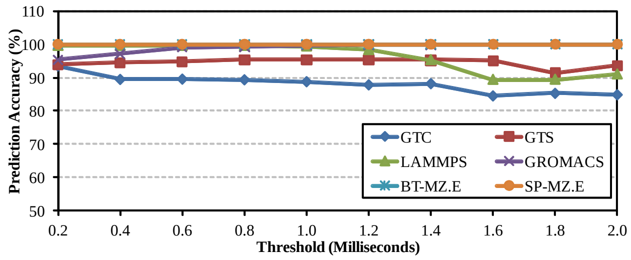


Figure 9. Sensitivity of prediction accuracy to the threshold value (Measured with 1536 cores on Hopper).

whereupon the scheduler enters the next step; otherwise, the signal handler returns and analytics process runs at full speed until the next scheduling point.

2) Identifying Contentious Analytics Processes: each GoldRush scheduler instance determines whether the local analytics process to which it belongs is contributing to interference. Toward that end, it uses the L2 Cache Miss Rate (L2 Cache Misses per Thousand Cycles) as the indicator for the analytics process' contentiousness. If this miss rate is greater than some threshold value, then the analytics process is subject to execution rate throttling. This is because an analytics process with high L2 Cache Miss Rate is likely to impose pressure on the shared L3 cache and on other shared resources, such as memory controllers and memory bus bandwidth.

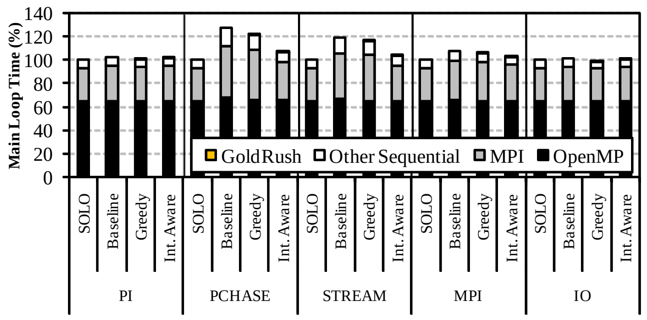
3) Throttling the Execution Rate of Analytics: the scheduler throttles an offending analytics process' execution rate by putting it to sleep for some short period of time, by calling the usleep() function. When the sleep duration is exceeded, the scheduler's signal handler returns. The analytics then runs at full speed until the signal handler is triggered again, repeating the three scheduling steps. Since sleep duration controls the amount of idle cycles not used by analytics, the duration's value along with the scheduling interval used jointly provide useful knobs for controlling the percentages of idle cycles being harvested.

3.5.2 Greedy Policy

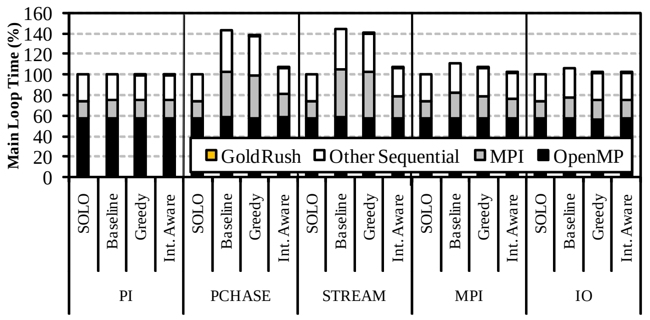
Under the Greedy policy, the analytics-side scheduler is disabled so that analytics processes run at full speed for all idle periods selected by the simulation-side prediction module. This policy differs from the OS baseline solution in that it relies on simulation-side prediction to filter out short idle periods. Comparing this Greedy policy with the Interference Aware policy and the baseline solution helps isolate the effects of simulation-side prediction and analytics-side scheduling.

3.6 Usage of GoldRush

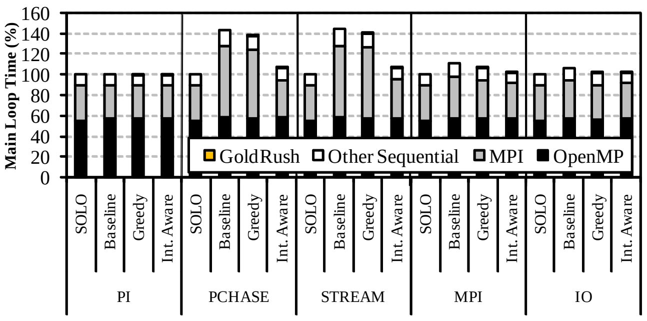
GoldRush makes it feasible to deploy in situ analytics onto compute nodes so that useful analytics can run on otherwise-wasted idle resources, close to the data source (simulation), and in parallel with the simulation. It can improve the performance and/or resource usage of scientific applications' online analytics and I/O pipelines. A sample usage of GoldRush is to run as much analytics work on idle resources as the idle capacity permits, so that the amount of dedicated resources (e.g., dedicated cores [6] or staging nodes [46]) for online analytics can be reduced or even avoided. Another usage is to perform data-reduction analytics operations with idle resources in compute nodes to reduce downstream data movements along the I/O pipeline.



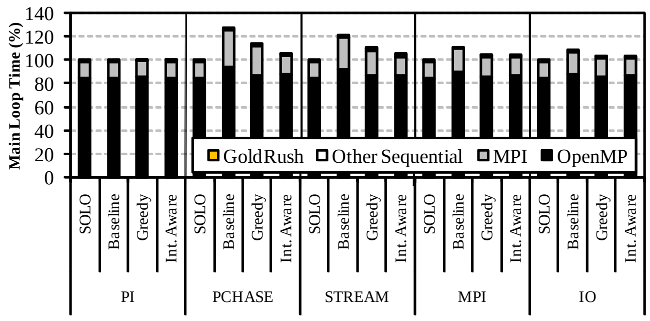
(a) GTC



(b) GTS



(c) GROMACS



(d) LAMMPS

Figure 10. Simulation performance with 1024 cores on Smoky cluster. The legend "GoldRush" refers to the time which the simulation spends in GoldRush operations (monitoring, prediction and signaling). Such overheads are very low (<0.3%).

4. PERFORMANCE EVALUATION

This section's experimental evaluations have three purposes: (i) analyze the cost and benefit of GoldRush runtime and its advantages over the OS baseline solution; (ii) measure the improvement of application performance and resource efficiency achieved by GoldRush for real-world applications; and (iii) assess the scalability of GoldRush with increasing machine size and node core count. The experiments are conducted on NERSC's Hopper Cray XE6 [10] and ORNL's Smoky cluster [34].

4.1 Benefits of Synergistic Scheduling

Our first set of experiments co-runs simulation with "unrelated" analytics (the analytics does not operate on simulation output but on its private data set) under different scheduling policies. Here we evaluate scenarios where there is interference between the simulation and analytics. Note those are less likely to occur with "related" analytics in which there is cache-friendly, constructive data sharing between simulation and analytics -- due to producer-consumer data reuse relationships. The purpose of these experiments is to assess GoldRush's ability to mitigate destructive interference between simulation and analytics. We co-run the four simulation codes (GTC, GTS, GROMACS and LAMMPS) with the five synthetic analytics benchmarks in Table 2. The simulation and analytics are set to run in four different configurations:

Case 1 (Simulation in Solo): Simulation is run without analytics; OpenMP worker threads do busy waiting in idle periods.

Case 2 (OS Baseline Solution): Simulation and analytics are co-located; OS schedules analytics processes to run whenever simulation's OpenMP worker threads yield CPUs.

Case 3 (Greedy Scheduling): Simulation-side GoldRush runtime selects idle periods, resumes and suspends analytics with signals; Analytics-side GoldRush scheduler is disabled.

Case 4 (Interference Aware Scheduling): Simulation-side GoldRush selects idle periods to run analytics, resumes and suspends analytics, and also records simulation main threads' IPC values in shared memory buffer during idle periods. Analytics-side GoldRush scheduler does interference detection and control.

Simulations and analytics are run with 1024 cores on the Smoky cluster. They are placed in compute nodes as shown in Figure 4.

4.1.1 Benefits of GoldRush

Figure 10 shows the simulation's main loop time in the four cases. GoldRush with its greedy policy can improve the performance of the four simulations over the OS baseline solution. This demonstrates the importance of selecting proper idle periods at the simulation side. GoldRush with its interference aware policy can further improve simulation performance over the greedy policy, resulting in 9.9% on average and up to 42% performance improvement over the OS baseline solution. Figure 10 shows that such improvements are due to the reduction of the "Main-Thread-Only" portion of the main loop time. The difference of simulation run time in solo vs. under interference aware scheduling is at most 9.1% (GROMACS running with PCHASE) and 1.7% on average among all test cases, meaning that the simulations' performance is close to the optimal. These results demonstrate that GoldRush's interference aware scheduling can mitigate potential interference effects between the simulation's main threads and analytics processes during idle periods. Such advantage is the most evident for memory intensive benchmarks like STREAM and PCHASE, as they cause severe contention on shared resources in memory hierarchy.

There is a trade-off between the amounts of idle cycles to harvest vs. the impact on simulation. Such tradeoff can be managed by tuning the parameters of scheduling policy (Section 3.5). In our tests, we conservatively set the idle period duration selection threshold to 1ms, scheduling interval to 1ms, IPC threshold to 1, L2 Miss Rate to 5, and sleep duration to 200µs. Such setup results

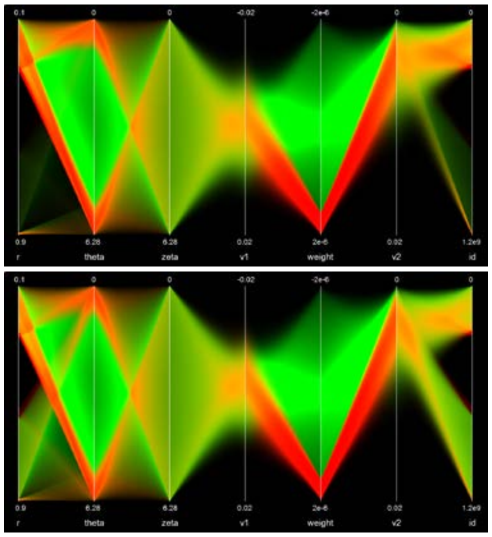


Figure 11. Parallel coordinates for GTS particle data. The two images are drawn from 2 timesteps of particle data each with 120GB in size. The red lines highlight particles with the absolute 20% largest weights.

in significant performance improvements at simulation side as shown in Figure 10, and meanwhile the aggregated amount of harvested idle periods is at least 34%, and 64% on average, of total available idle time. A thorough study of parameter tuning is left for future work.

4.1.2 Costs of GoldRush

The runtime cost of GoldRush at the simulation side can be quantified by the performance difference between GTS running in solo vs. co-running with analytics under the control of GoldRush. As mentioned earlier, this difference is 1.7% on average.

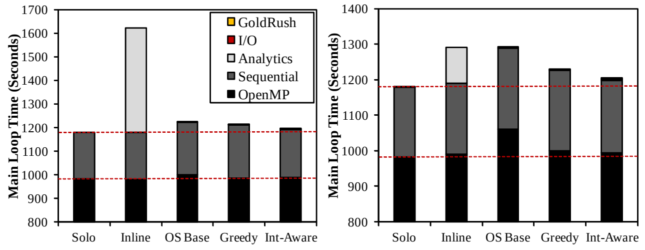
The simulation side cost of GoldRush can be further divided into two parts: the first part is the time spent in the GoldRush runtime itself (i.e., the time to execute the GoldRush marker APIs and monitoring signal handler), and the second part is simulation slowdown due to context switches and remaining interference from analytics. We internally instrument GoldRush and find that the aggregated time of the GoldRush runtime itself is small, constituting no more than 0.3% of the simulation's main loop time. Concerning runtime monitoring, the measured memory usage of storing GoldRush monitoring data in main memory is no more than 5KB per simulation process in all test cases.

4.2 GTS Application with In Situ Analytics

GTS (Gyrokinetic Tokamak Simulation) is a global three-dimensional Particle-In-Cell (PIC) code used to study the micro-turbulence and associated transport in magnetically confined fusion plasma of tokamak toroidal devices [41]. GTS outputs particle data during simulation. We apply GoldRush to manage GTS to co-run with two representative particle data analytics.

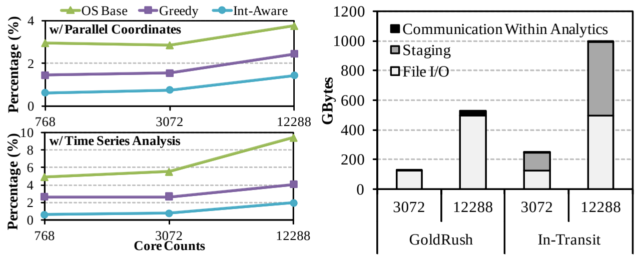
4.2.1 Parallel Coordinate Visual Analytics

Parallel coordinates is a visualization method commonly used to depict and analyze multivariate data [12][31]. We implement this method for GTS particle data. Each GTS particle has seven attribute, including coordinates, velocities, weight and particle ID. Each processor first generates its local plot of parallel coordinates from the selected particles. Then, all processors collectively



(a) w/ Parallel Coordinates **(b) w/ Time Series**

Figure 12. GTS performance with 12288 cores on Hopper.



(a) Scaling of GTS Slowdown **(b) Data Movement Costs**

Figure 13. Scaling results on Hopper. Figure 13 (a) shows the slowdown of GTS (comparing to Solo case) with different scheduling policies. Figure 13 (b) compares the data movement costs of running parallel coordinates in situ vs. in transit.

generate the final plot through parallel image compositing [44]. Multiple plots of parallel coordinates can be generated and composited to show the relationship between different groups of particles. Figure 11 shows the parallel coordinates for two time steps, where the green areas correspond to all particles, and the red areas corresponds to the particles with the absolute 20% largest weights. Our parallel coordinate analytics can clearly show the evolution of particle data distribution at large scale.

GTS is run with a typical setup, which results in particle data output size of 230MB per process. GTS outputs particle data every 20 iterations. Each GTS MPI process with 6 OpenMP threads is placed onto a separate socket on Hopper's 4-socket compute node. Weak scaling is applied to GTS from 768 to 12288 cores. Within each node, 20 visual analytics processes are placed onto the cores where the simulation's OpenMP threads are running. The 20 analytics processes are divided into 5 groups. Each group has 4 processes with one process running on a separate socket. GTS particles output data of successive timesteps are distributed among the 5 analytics process groups in a round-robin manner via the ADIOS shared memory transport [47]. Both the original particle data and the generated images are written to the file system.

For comparison, we also run GTS and visual analytics "*Inline*": the simulation directly calls the visual analytics routine. In this way, simulation and analytics are performed synchronously. We use a multi-threaded OpenMP version of the parallel coordinates processing routine to get the best possible inline performance.

Performance: Figure 12 (a) shows the main loop time of GTS simulation with 12288 cores on Hopper. Similar to previous experiments, the performance of GTS is best with GoldRush interference-aware scheduling. "Inline" has worst performance, due to synchronously performing analytics and file I/O. Figure 13 (a) shows the scaling of simulation-side slowdown. The GoldRush interference aware policy has better scalability than the OS baseline solution, which promises its utility at even larger scales.

Cost I (CPU Hours): with the same number of compute nodes used, using GoldRush leads to the least usage of CPU Hours.

Cost II (Data Movement Volumes): an alternative to co-locating simulation and analytics is to perform analytics "*In-Transit*": additional compute nodes are allocated to host analytics; data is moved from the simulation to analytics through the RDMA-based data staging transport in ADIOS I/O library [1]. This makes it possible to avoid contention on compute nodes, but results in additional data movement across the interconnect (which can also introduce perturbation to simulation [1]). Figure 13 (b) compares the data movement volumes under the GoldRush vs. In-Transit setups, where a 1:128 ratio of compute to staging nodes is used. We note that placing analytics onto a smaller number of staging nodes reduces MPI communication cost within the parallel coordinates analytics (for the image composition), but doing so adds data movements between the simulation and analytics (i.e., the staging traffic). Since placing analytics within the compute node and using GoldRush to schedule its execution can already achieve close-to-optimal performance, it is more efficient to use GoldRush rather than In-Transit for this GTS analytics use case. More generally, of course, In-Transit solutions remain important, because one must "size" on-compute-node analytics to match available idle resources. We leave the creation of general methods for such sizing to future work.

4.2.2 *Time Series Analytics*

Time series analysis [27] is essential for understanding particle temporal behavior. A basic operation of time series analysis is to iteratively access the data of each particle in the arrays of different time steps. A common data access pattern can be simply represented as $A[ti][p] = f(B[ti][p], B[ti+1][p])$ for two time steps, where, for a particle p, A is a derived variable whose value at the time step ti depends on the original variable B at ti ad ti+1. For example, the displacement of a particle is computed from its positions at two time steps. However, we note that it is non-trivial to generate the particle trajectories in parallel [27][29], which is out of the scope of this paper. In our study, we assume that we already have the time-series data of each particle and emulate the data access pattern with a synthetic code.

We co-run the code that exercises the data access pattern on GTS particle output data. Due to its streaming access pattern, the time series analytics causes 15.2 L2 cache misses per thousand instructions on Hopper. As shown in Figure 12 (b) and 13 (a), this results in up to 9.4% slowdown of the GTS simulation with 12288 cores under the OS scheduler. The GoldRush interference aware scheduler reduces such interference to at most 1.9% and manages to complete all analytics processing with available idle resources.

4.3 Varying Architecture - Intel Westmere

In order to evaluate the effectiveness of GoldRush across different architectures and its scalability within a node, we conduct experiments on a 32-core Intel Westmere machine. The machine has 4 sockets each with 8 cores at 2.13GHz, with a 32KB D-Cache, 32KB I-Cache, and a 256KB inclusive unified L2 Cache. All 8 cores within a socket share a 24MB inclusive unified L3 Cache. Each of the 4 sockets belongs to one NUMA memory domain with 32GB DDR3 memory in each domain. We run GTS with 4 MPI processes and 8 threads per process on this machine.

Figure 14 (a) shows the main loop time of the GTS simulation co-running with parallel coordinate analytics. The simulation's

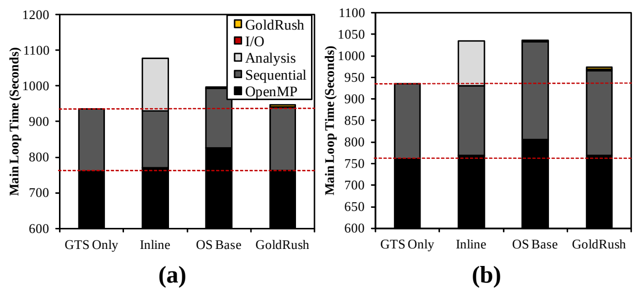


Figure 14. Simulation and analytics execution timeline.

OpenMP time increases by up to 5% under OS scheduler. This is because the OS scheduler does not entirely suspend analytics and thus, incurs unnecessary scheduling overhead. GoldRush with its greedy policy, however, results in GTS performance within 99% of the optimal. The less than 1% performance loss is due to time spent in the shared memory transport and the GoldRush runtime.

When co-running GTS with the contentious time series analytics under OS baseline scheduling, GTS can be significantly slowed down (up to 11%), as shown in Figure 14 (b). On the other hand, with the interference aware GoldRush scheduling, interference is again greatly reduced. This, together with previous results, demonstrates GoldRush's ability to mitigate interference between co-running simulation and analytics across different architectures.

5. RELATED WORK

In Situ Scientific Data Analytics. In situ data analytics and visualization has gained much recent attention from the HPC community. Current work falls into two categories: (i) data analytics and visualization algorithms, such as in situ indexing [43], compression [14], feature extraction [39], and visualization techniques [2][42], and (ii) supporting platforms. Regarding the first category, the GoldRush system can be readily used to run various data analytics with idle resources in compute nodes for resource-efficient, near-source data processing. As to the second category, systems like PreDatA [46], GLEAN [39], NESSIE [25] and DataSpaces [5], all support In-Transit data processing (i.e., deploy data analytics on auxiliary nodes and move data from simulation to analytics across interconnect), which is orthogonal to our work. One attractive usage of GoldRush is to run data-reducing operations on compute nodes to filter or pre-process simulation output data before sending data to In-Transit processing nodes. Finally, Damaris [6] and Functional Partitioning [15] use dedicated cores on compute nodes for file I/O and other data operations; such solutions are easily realized with GoldRush.

Cycle Stealing. Idle CPU cycles pervasively exist on PCs and servers. There has been extensive work on leveraging unused idle cycles for useful computation. Examples include Condor [17], BOINC [3], and other volunteer computing systems. To the best of our knowledge, GoldRush is the first system to harvest fine-grained idle cycles from large-scale scientific simulations on HEC platforms for online data analytics. Linger Longer [32] shares similarity with our work since it enables aggressive resource sharing between host applications and background jobs. GoldRush differs in its demonstrated scalability and in its ability to control interference for tightly synchronized host applications (parallel simulations).

Contention Mitigation on Multi-Core Platforms. Resource contention has been recognized as a severe performance issue for consolidated workloads on multi-core platforms. Software solutions to this problem include thread mapping [48] and scheduling [15], cache partitioning [18], and compiler-time code transformation for cache behavior optimization [33][36]. We borrow from such work to implement a special case for contention aware scheduling in which analytics processes detect contention with high-priority simulation and dynamically back off to mitigate interference. Most similar to our work are CAER [21] and ReQoS [37], both of which target data center applications.

Optimizing MPI/OpenMP Hybrid Programs. There has been previous work on tuning performance and/or power efficiency for MPI/OpenMP hybrid codes. One approach is to overlap sequential code regions with parallel OpenMP regions [30], but its applicability is application-specific, constrained by data and control dependencies as well as by thread safety in MPI libraries. Another approach by Li et al. [18] applies Dynamic Concurrency Throttling and Dynamic Voltage and Frequency Scaling to OpenMP parallel phases, for power savings. The slack prediction used in that approach is akin to GoldRush's idle period prediction: it estimates the difference in the duration of OpenMP parallel phases between the non-critical vs. critical (the slowest) MPI processes (i.e., the slack time), so that the non-critical processes can be run with reduced CPU frequencies during slack time. The duration of a sequential period between two successive OpenMP parallel phases is measured directly and used as input parameters to slack prediction. GoldRush, instead, dynamically predicts the duration of those sequential periods within each MPI process; and its purpose is not to reduce the power consumption of a simulation code running in solo, but to orchestrate the execution of coupled simulation and analytics to improve their overall performance and resource efficiency. It would be interesting, however, for a MPI/OpenMP hybrid simulation code, to use both methods: to optimize power efficiency of the OpenMP parallel phases and to apply GoldRush to schedule in situ analytics during idle periods outside the OpenMP phases. Also interesting to GoldRush is to leverage Li's work on dynamically varying OpenMP thread count for simulation's OpenMP phases: this may help yield even more idle resources for in situ analytics.

6. CONCLUSIONS AND FUTURE WORK

In situ scientific data analytics has been gaining wide adoption by scientific applications to cope with the severe I/O bottleneck on HEC platforms. This paper makes several key contributions to improving the online execution of data analytics. We first show that even leadership simulations leave considerable compute node resources unused. This is not because such codes are ill-tuned or configured, but because many such unused resources often occur as modestly sized idle periods not easily utilized by the dense core methods constituting the bulk of a typical simulation's computation. Unfortunately, this fact also makes such idle periods difficult to use for analytics. This is the key challenge addressed by the GoldRush system developed in our work. GoldRush applies fine-grained scheduling to "steal" idle resources from simulation in ways that incur negligible runtime overheads and minimize interference between the simulation and analytics. Key to its effectiveness are (i) judiciously selecting appropriate idle periods based on online performance monitoring and prediction, and (ii) dynamically detecting interference and mitigating it by throttling analytics' execution. Experiments with representative applications at large scales (up to 12288 cores) and on different architectures show that resources harvested on compute nodes can be used to perform useful analytics, significantly improving resource efficiency, reducing data movement costs incurred by alternate solutions, and posing negligible impact on simulations.

There are several directions for future work. First, we plan to develop automated resource provisioning methods, on top of GoldRush, to properly "size" the amount of analytics co-located with the simulation. Second, we will consider the challenges posed by graph-based analytics, which will likely be more disruptive to co-running simulations than the analytics used in this paper. Third, the prediction of idle period duration will benefit from the use of rigorous forecasting methods, particularly for simulations with more irregular and dynamic performance characteristics (e.g., AMR codes).

7. ACKNOWLEGEMENTS

We thank Stéphane Ethier for his help with the GTS application and visual analytics. We also thank the anonymous reviewers and our shepherd, Frank Mueller, for their insightful feedback. This work was funded by Scientific Data Management Center, U.S. Department of Energy, and Center for Exascale Simulation of Combustion in Turbulence, U.S. Department of Energy.

8. REFERENCES

[1] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan, and S. Klasky, Just in time: adding value to the i/o pipelines of high performance applications with jitstaging, In *HPDC*, 2011.

[2] J. C. Bennett, H. Abbasi, P. Bremer, R. Grout, A. Gyulassy, T. Jin, et al. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *SC*, 2012.

[3] BOINC. Open-source software for volunteer computing and grid computing. http://boinc.berkeley.edu/. 2013.

[4] Cray Inc. CrayPat Performance Analysis Tool. http://docs.cray.com/. 2013.

[5] C. Docan, M. Parashar, S. Klasky. DataSpaces: an interaction and coordination framework for coupled simulation workflows. In *HPDC*, 2010.

[6] M. Dorier, Using dedicated i/o cores for scalable post-petascale hpc simulations. In *ICS*, 2011.

[7] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Geveci, M. Rasquin, K. E. Jansen, The paraview coprocessing library: a scalable, general purpose in situ visualization library. In *LDAV*, 2011.

[8] GROMACS. http://www.gromacs.org/. 2013.

[9] E. R. Hawkes, R. Sankaran, and J. H. Chen, Direct numerical simulation of turbulent combustion: fundamental insights towards predictive models. In *Journal of Physics: Conference Series*, 2005, pp. 65-79.

[10] Hopper Cray XE6 at NERSC. http://www.nersc.gov/systems/hopper-cray-xe6/, 2013.

[11] T. Hoefler, T. Schneider and A. Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *SC*, 2010.

[12] C. Jones, K.-L. Ma, S. Ethier, W.-L. Lee. An ontegrated exploration approach to visualizing multivariate particle data. In *Computing in Science & Engineering*. Volume 10, Number 4, July/August, 2008, pp. 20-29.

[13] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R.

Samtaney, Grid-based parallel data streaming implemented for the gyrokinetic toroidal code. In *SC*, 2003.

[14] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, N. F. Samatova. Compressing the incompressible with ISABELA: in-situ reduction of spatio-temporal data. In *Euro-Par*, 2011.

[15] M. Lee, K. Schwan. Region scheduling: efficiently using the cache architectures via page-level affinity. In *ASPLOS*, 2012.

[16] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, G. Shipman. Functional partitioning to optimize end-to-end performance on many-core architectures. In *SC*, 2010.

[17] M. J. Litzkow, M. Livny, M. W. Mutka. Condor-a hunter of idle workstations. In *ICDCS*, 1988.

[18] D. Li, B. Supinski, M. Schulz, D. Nikolopoulos, K. Cameron. Hybrid mpi/openmp power-aware computing. In *IPDPS*, 2010.

[19] J. F. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rick i/o methods for portable high performance i/o. In *IPDPS*, 2009.

[20] Q. Lu, J. Lin, X. Ding, Z. Zhang, X. Zhang, P. Sadayappan. Soft-OLP: improving hardware cache performance through software-controlled object-level partitioning. In *PACT*, 2009.

[21] J. Mars, N. Vachharajani, R. Hundt, M. L. Soffa: Contention aware execution: online contention detection and response. In *CGO*, 2010.

[22] B. Miller, A. Bernat. Anywhere, any time binary instrumentation, In *PASTE*, 2011.

[23] B. Mohr, A. D. Malony, S. Shende, F. Wolf. Design and prototype of a performance tool interface for openmp. In *LACSI*, 2001.

[24] NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html. 2013.

[25] R. Oldfield, G. Sjaardema, J. F. Lofstead, T. Kordenborck. Trilinos i/o support (trios). In *Scientific Programming*, August 2012.

[26] PAPI: Performance Application Programming Interface, http://icl.cs.utk.edu/papi/, 2013.

[27] T. Peterka, R. Ross, B. Nouanesengsey, T.-Y. Le, H.-W. Shen, W. Kendall, J. Huang. A study of parallel particle tracing for steady-state and time-varying flow fields. In *IPDPS*, 2011.

[28] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics, In J Comp Phys, 117, 1-19 (1995).

[29] D. Pugmire, H. Childs, C. Garth, S. Ahern, G. Weber. Scalable computation of streamlines on very large datasets. In *SC*, 2009.

[30] R. Rolf, G. Hager, G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *PDP*, 2009.

[31] O. Rubel, Prabhat, K. Wu, H. Childs, J. Meredith, C.G.R. Geddes, E. Cormier-Michel, S. Ahern, G.H. Weber, P. Messmer, H. Hagen, B. Hamann, E.W. Bethel. High performance multivariate visual data exploration for extremely large data. In *SC*, 2008.

[32] K. D. Ryu, J. K. Hollingsworth. Linger longer: fine-grain cycle stealing for networks of workstations. In *SC*, 1998.

[33] A. Sandberg, D. Eklov, E. Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *SC*, 2010.

[34] Smoky Cluster. http://www.olcf.ornl.gov/computing-resources/smoky/, 2013.

[35] R. Stevens, A. White, et al. Architectures and technology for extreme scale computing. Technical report, ASCR Scientific Grand Challenges Workshop Series, December 2009.

[36] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: mitigating contention for qos in warehouse scale computers. In *CGO*, 2012.

[37] L. Tang, J. Mars, W. Wang, T. Dey, M. L. Soffa: ReQoS: reactive static/dynamic compilation for qos in warehouse scale computers. In *ASPLOS*, 2013.

[38] Vampir Performance Tool. http://www.vampir.eu/. 2013.

[39] V. Vishwanath, M. Hereld, M. E. Papka, Toward simulation-time data analysis and i/o acceleration on leadership-class systems. In *LDAV*, 2011.

[40] V. Vishwanath, M. Hereld, V. Morozov, M. E. Papka. Topology-aware data movement and staging for i/o acceleration on blue gene/p supercomputing systems. In *SC*, 2011.

[41] W. X. Wang, Z. Lin, W. M. Tang, W. W. Lee, S. Ethier, J. L. V. Lewandowski, G. Rewoldt, T. S. Hahm, J. Manickam, Gyro-kinetic simulation of global trubulent tranport properties in tokamak experiments. In *Physics of Plasmas*, 2006, pp 59-64.

[42] H. Yu, C. Wang, R. W. Grout, J. H. Chen, K. Ma, In-situ visualizaiton for large-scale combustion simulations. In *CGA*, 2010.

[43] K. Wu, S. Ahern, E.W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, et al. FastBit: interactively searching massive data. In *SciDAC, Journal of Physics: Conference Series*, 2009.

[44] H. Yu, C. Wang, K.-L. Ma. Parallel volume rendering using 2-3 swap image compositing for an arbitrary number of processors. In *SC*, 2008.

[45] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki and H. Abbasi. Enabling in-situ execution of coupled scientific workflow on multi-core platform. In *IPDPS*, 2012.

[46] F. Zheng, H. Abbasi, C. Docan, J. F. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, M. Wolf, Predata-preparatory data analytics on peta-scale machines. In *IPDPS*, 2010.

[47] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.-A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, H. Yu. FlexIO: i/o middleware for location-flexible scientific data analytics. In *IPDPS*, 2013.

[48] S. Zhuravlev, S. Blagodurov, A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, 2010.