2013 42nd International Conference on Parallel Processing

Event Streaming for Online Performance Measurements Reduction

Jean-Baptiste Besnard*† jean-baptiste.besnard@cea.fr

Marc Pérache *† marc.perache@cea.fr

William Jalby † william.jalby@prism.uvsq.fr

*CEA, DAM, DIF
Bruyères-le-Châtel
F 91297 Arpajon, France

†Université de Versailles Saint-Quentin
45 Avenue des États-Unis
Versailles, France

***Abstract*—As the power of supercomputers is exponentially increasing, programmers are facing complex codes designed to comply with today's challenging architectural constraints. In such context, the use of tools within the development cycle, is becoming crucial in order to optimise applications at scale. However, it is not possible to obtain all measurements one can think of, because of the cost to produce, store and analyse large amounts of instrumentation-data. Moreover, the file-system is becoming a critical resource, subject to performance and even stability problems under load. This emphasises the need for an alternative approach to trace data management. This paper proposes an alternative to trace-based coupling between instrumentation and analysis. We present a distributed analysis engine, providing concurrent application profiling, thanks to runtime coupling. After demonstrating the advantages of this method in terms of parallelism, we present performance results and sample outputs for NAS-MPI benchmarks and a representative C++ MPI application.**

***Keywords*-Performance tools; Online trace analysis; MPI virtualization; Code coupling**

I. Introduction

With supercomputers gathering millions of cores, programmers must deal with the increasing complexity of their codes. Nowadays, massively parallel programs have to comply with a growing amount of constraints, arising from recent evolutions such as architectures' hybridisation, and from the vast increase in the number of cores. This complexity of the underlying architecture prevents performance from being anticipated. On the opposite, the process of porting legacy codes or developing new ones necessarily is an exploratory process guided by measurements. The need to design efficient codes on current and upcoming supercomputers makes the use of performance-tools compulsory.

Tools scalability is required to qualify real test cases related to a representative behaviour. However, in order to thoroughly comprehend the behaviour of a program, a large amount of data must be collected. This possibly leads to an important file-system stress — shared resource which already shows its limitations on peta-scale machines, with problems of meta-data server contention. In order to address this data-management problem, various strategies have been designed including tracing through parallel IO libraries [1]–[4] or embedded analysis [5]–[7] or a mix of both [8]–[10]. These approaches generally express a trade-off between execution perturbation and collected data accuracy. Limiting perturbation generally implies restraining measurements by filtering or locally reducing data. This paper proposes an alternative to this storage–reduction dilemma by completely removing traces and their associated IO-bottleneck while keeping fine-grained events. To do so, our approach consists in coupling instrumentation and analysis at runtime, thus, avoiding storage of temporary trace data without impeding programs with embedded analyses.

This work aims at developing a centralised performance measurement service which will provide analysis of concurrent programs and will simultaneously generate profiling reports while collecting global performance metrics. To do so, this paper presents both an online coupling mechanism for performance data offloading and the architecture of our distributed analysis engine as well as sample profiling outputs. This work also demonstrates the analysis of concurrent programs can be done in a single report, to our knowledge it is an original work which, for example, simplifies MPMD application analysis. Current implementation does not completely achieve the goal of a machine wide profiling engine. Reasons for this are discussed in Section III-C, but this work sets-up all the basis for a further implementation providing profiling as a service.

II. Contribution

This paper presents a new instrumentation–analysis coupling method. It provides end-to-end parallelism while maximising the bisection bandwidth by directly feeding the analyser without relying on a file-based trace. We also introduce a distributed analysis engine able to run multiple concurrent profiles, thanks to runtime coupling and a parallel data-centric task engine.

A. Online Coupling

As shown in Figure 1, when coupling instrumentation and analysis through file-system, a loss of parallelism necessarily appears because of the encapsulation required by parallel IO libraries trying to limit the file-system stress

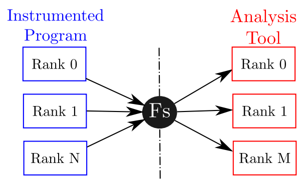

Figure 1. Classical instrumentation to file-based trace.

through a reduction of the number of files. Consequently, upon reading, data have to be explicitly redistributed to each analysing process, supposing that the trace is self-coherent with global identifiers. Moreover, processing must be done after trace completion, thus, preventing analysis to overlap with instrumentation. Analysing a trace generally consists in reducing performance data to express them as intelligible metrics. These reductions can take place within the application itself to limit the size of the trace, possibly impacting its performance and adding new entries in the trace format. Other types of analysis such as pattern detection in communications [11] which requires an inter-processes context, can benefit from being delayed to post-mortem — forcing temporary storage of unreduced data. Various strategies have been developed within the tools to overcome file-system limitations (see Section V): reducing collected data while providing pertinent measurement. This work provides an alternative to embedded performance events reduction by delegating the processing to an on-line analyser which can perform distributed analyses of arbitrary complexity without suffering from IO-bottleneck, mitigating performance impact.

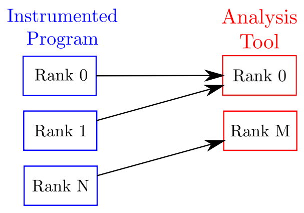

Figure 2. Instrumentation through runtime coupling.

Figure 2 presents our instrumentation–analysis runtime coupling through network data-streams. It simplifies data management and provides several performance improvements as a parallel IO library is no longer required. Data from the producer are directly forwarded to the consumer without any encapsulation. Besides, trace-format can be simplified if process mapping is surjective. In this configuration, each analysis process is mapped to a fixed set of instrumented programs, partially relaxing the need for global identifiers. By not using file-system, temporary trace data are not stored at all and do not have to be stored on a slower medium. Moreover, performance measurements are directly sent to the analyser, avoiding going back and forth between file-system servers — halving the overall network traffic. Streaming performance data also opens new parallelism opportunities such as pipe-lined and concurrent analyses which are not feasible with traces.

B. Analysis Engine

The analysis engine has been developed to provide multi-program profiling, relying on a runtime coupling with each instrumented program. It factors profiling resources and takes advantage of the pipe-lined parallelism implied by our coupling method. Our analyser is implemented as a parallel data-centric task engine. Its structure is inspired from Blackboards Systems [12]–[14].

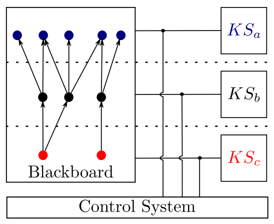

Figure 3. Canonical architecture of a Blackboard System.

Blackboard Systems are multi-agent architecture originally designed to build complex expert systems such as the Hearsay-II [15] for speech recognition or the HASP [16] system for continuous detection of submarines through hydrophones arrays. This kind of signal processing tasks shares a common need for multiple levels of interpretation. At each each step, a different kind of analysis and data representation is required as the information gains both in structure and semantic. Blackboard systems are named after the analogy of multiple experts or Knowledge Sources (*KS*) gathered around a blackboard (*BB*) and sharing their thoughts about a given problem while taking opportunity of valuable data becoming available to propose solutions. To take into account the scheduling process for a computer based implementation, a control component is added in order to decide which KS is particularly suited, based on an heuristic of its possible contribution, to process a given data. Figure 3 presents the canonical architecture of a blackboard system with Knowledge Sources working at a given abstraction level over semantically linked data.

Blackboard systems provide an anonymous storage structure. It allows cohabitation for chained analyses which are dedicated to a set of data types. Their processing are triggered each time suitable data are pushed on the blackboard. Our implementation relies on several worker threads, providing analysis with natural parallelism. Figure 4 presents a sample data-flow analysis implementation on a blackboard: event packs streamed from the instrumented application are 'pushed' on the blackboard, which triggers their unpacking by the 'KS Unpacker' which in turn pushes all the individual

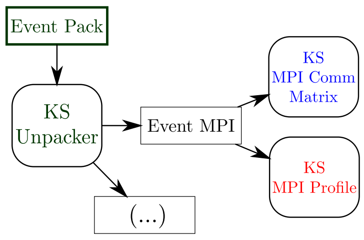

Figure 4. Example of data-flow analysis implemented in a Blackboard.

events. Then, MPI events are processed by both topological analysis and MPI profiler, in order to reduce events to their individual data-structure. This approach does not only simplify data-analysis expression, but also enables to analyse straightforward concurrent programs. At data-flow level, processing simply has to be replicated for each program, using multi-level blackboard, each level being dedicated to an application. As shown in Figure 5, a new KS in charge of dispatching each event pack to its associated blackboard level, provides a direct multi-instrumentation support.

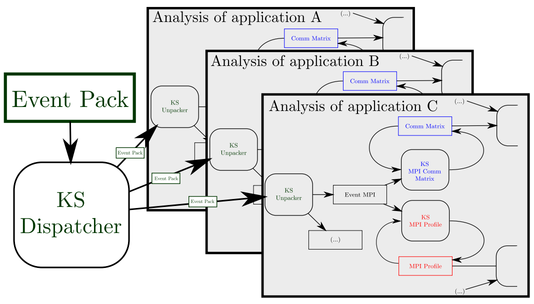

Figure 5. Multi-level blackboard allowing concurrent application profiling.

Knowledge sources can be developed in separated shared libraries which can be loaded dynamically, integrating new KSs on the blackboard. This simplifies the development of profiling modules as they can be developed in a very orthogonal manner. Moreover, thanks to the shared library mechanism, these KS can rely on third party dependencies such as image processing libraries and can perform analysis of arbitrary complexity with various levels of integration on the Blackboard. Modules can just refer to a single event for notification purpose or completely integrate themselves on the blackboard with multiple KS and data-types in order to benefit from data-flow parallelism. As discussed in Section III-B and IV-D, our analysis engine relies on a Pthread based blackboard and implements multiple MPI related analysis tested on both Tera 100 [17] and Curie [18] supercomputers. Thanks to on-line event reduction, a user launching multiple instrumented applications is able to get a dedicated report with full details of each programs behaviour, briefly after execution ends.

III. Implementation

This section presents the implementation of our online measurement reduction framework. We first introduce our coupling method, detailing how application virtualization, mapping and coupling are achieved in the context of multi-instrumentation. Then, we present our parallel Blackboard architecture and data management methods.

A. Online Coupling

In order to implement an on-line coupling method, three conditions must be satisfied :

- **Transparent cohabitation:** two programs must run concurrently, in accordance with the scheduler.
- **Mapping:** groups of processes must connect to each other.
- **Communication:** each process has to efficiently communicate, using a persistent asynchronous data stream.

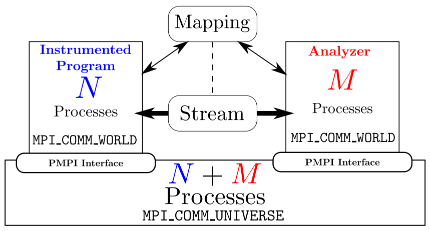

Figure 6. Overview of the runtime coupling mechanism.

As presented in Figure 6, these three requirements are satisfied in our implementation using MPI in MPMD mode. In this purpose, we provide transparent cohabitation with *MPI virtualization*, mapping with the *VMPI_Map* primitive and asynchronous communication with *VMPI_Streams*.

Virtualization is achieved using a simple mechanism similar to the one implemented in P^{N}MPI's virtual module [19]. It consists of replacing every references to `MPI_COMM_WORLD` by a reference to a sub-communicator. This is done by intercepting every MPI calls through the PMPI interface. Originally implemented over P^{N}MPI, we had to rewrite our own virtualization outside of P^{N}MPI to provide an integrated library which can be preloaded on MPI programs without code modification, recompilation or binary patch. Moreover, providing module's interface to the host application was not convenient as our library was divided in two, separating streams and virtualization. Therefore, we decided to implement our library in a single package which can be easily linked or preloaded in order to virtualize a program. To do so, we wrote in C a MPI wrapper generator, very similar features as P^{N}MPI's python one, with some extra options such as conditionals. Using this wrapper, we are able to generate a complete virtualization interface directly compiled into the VMPI library. Thanks to its extended interface, this wrapper was also used to generate the PMPI interface used by our instrumentation library. When running virtualized, each program runs transparently, in its own `MPI_COMM_WORLD` whereas the real `MPI_COMM_WORLD` is still available to perform inter-application communications as `MPI_COMM_UNIVERSE` (see Figure 6). Processes are grouped in partitions either by names or command lines. Partition descriptions are available within each process and can be queried by name, allowing opportunistic mapping those partitions offering various services.

A basic component called VMPI_Map is provided in order to simplify process to process *mapping*. It can be used to generate a mapping between two partitions by assigning to each process a set of matching processes located in the remote partition according to a given policy.

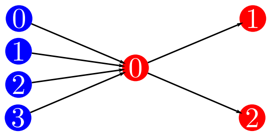

Figure 7. Mapping of two partitions with a pivot.

As shown in Figure 7, when mapping two partitions, the larger partition becomes the slave and the smaller one the master. Each process from the slave partition sends its global rank to the root of master partition (available through partition descriptions). Then, each time the master partition's root receives a rank, it picks up a local rank within its partition (including itself) according to a predefined policy, and associated local and remote ranks both-ways. In some cases, centralised mapping can be avoided when topologies can be computed locally (for example topologies a and c of Figure 8). But, for specific cases, this approach allows more general mappings by providing a user-defined function which takes a source as a parameter and returns the target. Moreover, it simplifies synchronisation when handling complex topologies by providing a pivot which broadcasts the end of the mapping to every process.

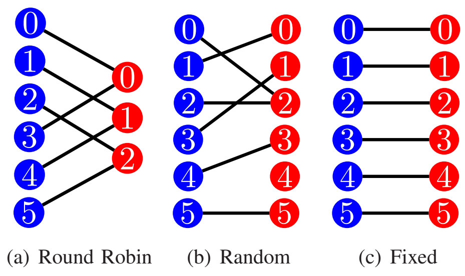

Figure 8. Illustration of the three default mapping topologies.

The three default mappings are detailed in Figure 8. In order to be valid for code coupling purposes, each process has to be associated with at least one other process. Partial mappings, more advanced than the trivial random one, are nonetheless possible through user-defined mapping-functions. Moreover, a VMPI_Map can be filled in an additive manner: a partition can compute its mapping to several other partitions by successively appending new entries. This feature which is particularly useful for multi-instrumentation.

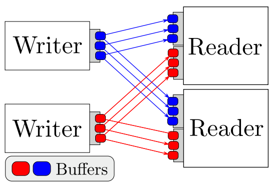

Figure 9. Architecture of a VMPI_Stream.

Efficient streamed-communications between partitions are provided by *VMPI_Streams* which are persistent asynchronous communication channels. They can be either multi- or uni-directional. They provide an interface and behaviour close to UNIX pipes. Writing to a stream is then non-blocking, until all asynchronous buffers are full, preserving an adaptation window between data producers and consumers. As depicted in Figure 9, each stream allocates buffers at both read and write endpoints to provide asynchronous streaming. Note that read endpoints have N_A (with $N_A = 3$ in our example) buffers per incoming stream to ensure that there is always a buffer available to receive any incoming data. This allows asynchronous reception of data blocks and without unexpected message as the MPI runtime is able to write directly in the reception buffer. On the opposite, on the writer side, N_A output buffers are shared between multiple endpoints, primarily to limit memory footprint (when using streams for instrumentation purpose, block size tends to be large ≈ 1 MB). VMPI_Streams can be created from a mapping in order to link two or more partitions, they can also be used between two arbitrary ranks. As a single stream can be connected to multiple endpoints, streams are initialised with a load-balancing policy which can be different at the two endpoints. Three basic policies are proposed : none, random, round-robin. Non blocking read is also supported to avoid deadlocks in multiple endpoints mode. When set, the call returns `EAGAIN` and tries the next endpoint according to the policy, avoiding circular waits.

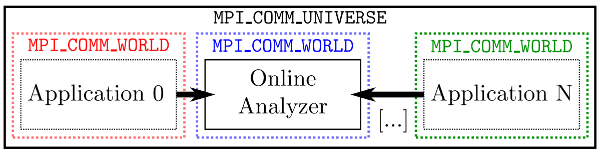

Figure 10. Runtime coupling for multi-instrumentation purpose.

The combination of virtualization, mappings and streams we briefly exposed, provides all the necessary components to perform runtime coupling. In order to illustrate our method, we perform the dynamic mapping shown in Figure 10. It consists in mapping N partitions to one — mapping strictly identical to the one we do when instrumenting multiple applications. The two source codes of Figure 11 and 12 are sufficient to build the runtime-coupling described in Figure 10. In this case, each application is transparently running in its sand-boxed communicator thanks to virtualization. They are able to connect to each other using our mapping component and to setup inter-application communication channels which will adapt to available resources thanks to VMPI_Streams's load-balancing support. This example demonstrates that our approach efficiently handles communications and mapping from N to one partitions making multi-instrumentation trivial.

```
MPI_Init( &argc , &argv );
/* Fill in mapping data */
VMPI_Map map;
VMPI_Map_clear( &map );
/* Retrieve analyzer partition */
VMPI_Partition_desc *p_an =
                    VMPI_Get_desc_by_name("Analyzer");
/* Could not find analyzer */
if(!p_an ){
  printf("Could not locate analyzer partition\n");
  exit(1);
}
/* Map to analyzer */
VMPI_Map_partitions( p_an->id ,
                    VMPI_MAP_ROUND_ROBIN, &map );
/* Setup Stream */
VMPI_Stream st;
/* Initialize stream */
VMPI_Stream_init( &st , 1024*1024,
                 VMPI_STREAM_BALANCE_ROUND_ROBIN);
/* Create streams according to mapping */
VMPI_Stream_open_map( &st , &map, "w" );
void *buff = calloc( 1024 * 1024, 1 );
/* Send some data */
int i;
for( i = 0 ; i < 1024 ; i++ )
  VMPI_Stream_write( &st , buff , 1 );
/* Close Stream */
VMPI_Stream_close( &st );
free( buff );
MPI_Finalize();
```

Figure 11. Sample code for a runtime-coupled instrumented program.

B. Parallel Blackboard

In order to reduce the large amount of data coming from instrumented programs, our distributed analysis engine relies on a parallel blackboard. It provides modularity and allows dynamic loading of plugin as well as natural parallelism by describing the processing as a data-flow. Our blackboard implementation relies on two main components: data entries (DE) and Knowledge Sources (KS). A Data Entry can be defined as a tuple: $\{Type, Size, Payload\}$ with *Type* being an integer identifier, *Payload* an arbitrary blob of data the size of which is given by *Size*. A knowledge source is a couple: $\{\{Sensivities\}, Operation\}$ with *Sensitivities* being a set of $\{Types\}$ triggering an *Operation* defined as a function called over the input data. Moreover, a KS can have multiple sensitivities of the same type and is able to submit any type of data entry and register new KS.

```
/* Set partition name */
VMPI_Set_partition_name( "Analyzer" );
MPI_Init( &argc , &argv );
/* Fill in mapping data */
VMPI_Map map;
VMPI_Map_clear( &map );
int i, ret;
for( i = 0 ; i < VMPI_Get_partition_count(); i++){
  /* Map each partition except myself */
  if( i != VMPI_Get_partition_id() )
    VMPI_Map_partitions( i ,
                         VMPI_MAP_ROUND_ROBIN,&map );
}
/* Setup Stream */
VMPI_Stream st;
/* Initialize stream */
VMPI_Stream_init( &st , 1024*1024,
                 VMPI_STREAM_BALANCE_ROUND_ROBIN);
/* Create streams according to mapping */
VMPI_Stream_open_map( &st , &map, "r" );
/* Allocate input buffer */
void *buff = malloc( 1024 * 1024 );
/* Start Read Loop */
do{
  /* Read one block of 1M from stream */
  ret = VMPI_Stream_read( &st , buff ,
                          1, VMPI_STREAM_NONBLOCK);
  if( ret == VMPI_EAGAIN )
    continue;

  if( 0 < ret ){
    /* Process BUFFER */
  }
/* 0 means all remote streams are closed */
}while( ret != 0 );

VMPI_Stream_close( &st );
free( buff );
MPI_Finalize();
```

Figure 12. Sample code for a runtime-coupled analyser.

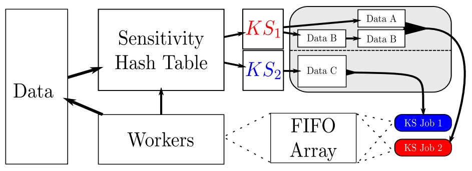

Figure 13. Architecture of the parallel blackboard.

In order to keep the control system as simple as possible, operations are first described as a static data-flow. A simplified form of opportunistic reasoning is provided by the ability of any KS to register or remove any KS including itself. The Control System (see Figure 3) is only in charge of triggering KS with satisfied sensitivities. Figure 13 presents the architecture of our parallel blackboard: when a data entry is submitted, matching sensitivities are looked up in the sensitivities hash table; if a matching KS is found, a reference to the data entry is pushed in a FIFO. In the case it is last unsatisfied sensitivity, a new job is created as a couple $\{\{Data\ entries\}, Operation\}$. In order to reduce contention, jobs are randomly pushed in an array of FIFOs, individually protected by a lock. A pool of workers is constantly looking for jobs by sweeping FIFOs from a random starting point while a back-off system prevents threads from spinning over the locks in the absence of jobs. In order to make parallelism manageable, data entries are generally read only and are managed using a ref-counting mechanism, a data being writable only if its ref-counter is equal to one. Besides, when data are pushed on the blackboard, they are stored in a buffer which is automatically freed after all the processings linked to this data are done. This allows the use of the blackboard as a temporary storage medium, freeing MPI_Streams's communication buffers and avoiding to block instrumented processes. The multi-level blackboard is implemented using data-entries identifiers which are computed as a hash of both level and data-type names. By doing so, identical KSs and data-types can be present in multiple blackboard levels, providing multi-analysis as depicted in Figure 5.

C. Implementation Limitations

As discussed in previous sections, our coupling framework is built over MPI in MPMD mode, despite allowing direct integration in existing supercomputers' batch managers and software stack, this design choice has drawbacks. Currently, a user who wants to instrument a set of programs using our tool has to launch a job consisting in his own programs plus the distributed analysis engine. Resources being statically assigned, if programs have very different wall-times, analysis resources will be over-sized for a part of the execution. A solution to overcome this problem would be MPI dynamic processes, but they are not particularly easy when spawning multiple communicating processes. Even if we deal with the complexity of MPI dynamic processes, we will not be able to provide an inter-node instrumentation service, because ideally, it should not be included in the batch manager but instead in an identified set of nodes providing a service. An implementation at network layer level would provide more flexibility with dynamic program registration, persistence of instrumentation servers, and more opportunities to manage authentication and centralisation of profiling metrics.

IV. Results

A. Test Platforms

All the tests of the following sections were done on Tera 100 and Curies Supercomputers. Tera 100 [17] has an aggregate peak performance of 1.2 PetaFlops. It hosts 140 000 cores in 4370 nodes with four eight core Nehalem EX at 2.27 GHz and 64 GB of memory. Tera 100's network is an Infiniband QDR network with a fat-tree topology. It runs an open-source software stack based on Linux and uses Slurm as batch manager. Curie [18] has the same architecture with an aggregate peak performance 1.36 PetaFlops in 5040 nodes with two eight core Sandy Bridge at 2.7 GHz and 64 GB of memory for a total of 80 640 cores.

B. Performance of VMPI_Streams

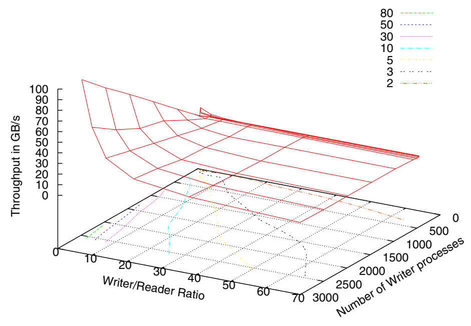

Figure 14. Global throughput of VMPI_Streams when writing 1GB per process at various $\frac{\text{writer}}{\text{reader}}$ ratios with programs of Figures 11 and 12.

Figure 14 shows the throughput which can be achieved on Tera 100 with the coupling codes of Figure 11 and 12 for different $\frac{\text{writer}}{\text{reader}}$ ratios. The number of reader N_r for a given number of writer N_w is computed as follows : $N_r = \lfloor \frac{N_w}{Ratio} \rfloor$ if $1 < \lfloor \frac{N_w}{Ratio} \rfloor$, with a default value of $N_r = 1$ to make sure there is always one process reading. As it could be expected the best throughput is obtained when there are as many readers than writers, with for example at 2560 writers and readers, a cumulative throughput of 98.5 GB/sec between the two partitions. This value has to be compared with the maximum IO throughput of Tera 100 which is of 500 GB/s for the whole machine which, scaled back to 2560 cores and considering an even bandwidth balancing (expected because of the fat-tree topology), gives a theoretical throughput of 9.1 GB/s. Consequently, at this scale, VMPI_Streams are competitive with the file-system approach until a ratio of one reader for $\approx$ 25 writers. Practically, ratios between $\frac{1}{1}$ and $\frac{1}{32}$ provide enough bandwidth for profiling purpose, $\frac{1}{10}$ being a good bandwidth–resource trade-off.

C. Instrumentation Overhead

Our instrumentation chain has been tested on Tera 100 on NAS-MPI Benchmarks (class C and D) and EulerMHD [20] which is a middle sized C++ MPI application which simulates Euler ideal magneto-hydrodynamic at high order on a 2D Cartesian mesh. All measurements were done three times and averaged, combinations of problem sizes and classes not supported by NAS benchmarks are omitted.

Figure 15 presents the relative overhead caused by our analysis tool (between MPI_Init and MPI_Finalize) when instrumenting MPI calls and their context with a $\frac{1}{1}$ ratio. This configuration maximises the bisection bandwidth and thus minimises the overhead. All overheads are all lower than 25% but vary with the application. In particular, NAS

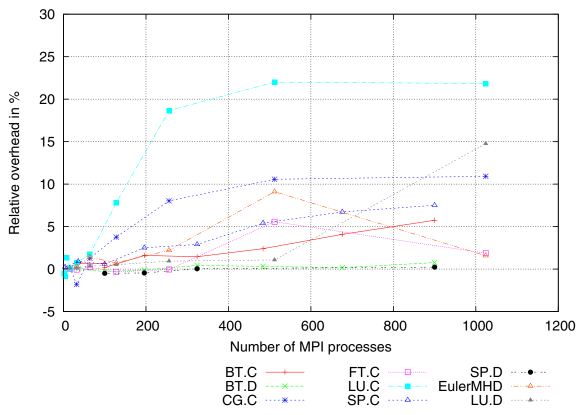

Figure 15. Relative overhead for NAS Benchmarks and EulerMHD running with one analysis core for one instrumented process (on Tera 100).

benchmarks in class C seem to expose a larger overhead than in class D. This behaviour can be understood by looking at the average instrumentation data bandwidth computed as $\overline{B_i} = \frac{\text{Total event size}}{\text{Execution time}}$. Applications with larger problem size, more time consuming in computation, issuing MPI calls less intensively, yielding a lower $\overline{B_i}$. For example, comparing SP.C and SP.D at 900 cores, we have $\overline{B_i}(\text{SP.C}) = 2.37$ GB/s and $\overline{B_i}(\text{SP.D}) = 334.99$ MB/s. Overhead is then correlated with the average instrumentation data bandwidth which has to fit in the available throughput (Figure 14) without impacting the instrumented program.

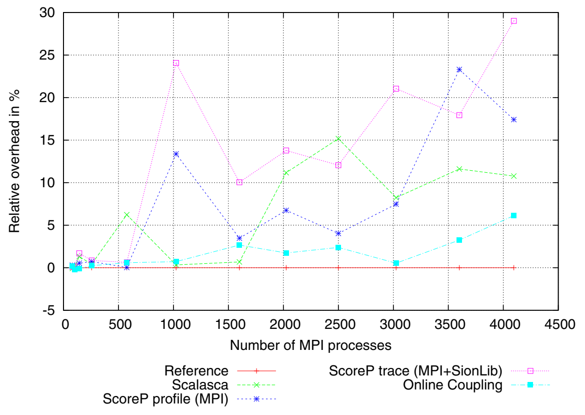

Figure 16. Relative overhead with different tools for NAS Benchmarks SP.D on the Curie supercomputer (averaged 5 times).

Figure 16 shows the relative overhead (between MPI_Init and MPI_Finalize) for NAS benchmark SP.D on the Curie [18] supercomputer, comparing our method with two profiling tools: Scalasca 1.4.3 [8], [21] and ScoreP 1.1.1 [22]. This last one generates either OTF2 traces (compatible with Vampir [23]) or runtime profiles for Scalasca or Tau. Measurements are done with default buffer configuration for MPI only (compiler instrumentation disabled), using SionLib [2] when generating ScoreP traces. It can be seen that on this benchmark, our online instrumentation has an overhead lower than file based traces despite manipulating larger volumes of data (ScoreP traces grow linearly from 313 MB to 116 GB and online coupling ones from 923.93 MB to 333.22 GB). This suggests that runtime-coupling is more scalable than the trace-based approach which might suffer from file-system limitations. We have no definitive explanation for the variations observed with other tools, they might be caused by varying machine load or process layouts (although nodes were allocated exclusively). Moreover, it shall be noted that NAS benchmarks have a reduced wall-time at larger scale, making them more subject to measurement noise despite several averaging passes.

D. *Sample Analysis Outputs*

This section gathers some extracts of profiling reports generated by our tool for NAS Benchmarks and EulerMHD. A profiling report is a latex document of 20 to 70 pages — depending on verbosity. The analyser supports different analysis producing various outputs such as topologies, profiles, temporal and spatial maps for MPI and POSIX calls.

Patterns from Figure 17 were generated by the topological module. This module generates communication graphs and matrices weighted in hits, total size and total time for every point to point communication types. Such matrix, weighted in total size for NAS Benchmark CG.D on 128 cores is shown in Figure 17(a) while Figure 17(b) presents its associated topology. Other topologies, also weighted in total size, are shown in Figure 17(c) for EulerMHD over 2048 cores and 17(d) for NAS Benchmark SP over 2025 cores. This kind of analyses is useful to express communications in space, emphasising local imbalances.

Another module generates density maps for process behaviour comparison. Such maps are available for all MPI and most POSIX calls in term of hits, time and total size (when suitable) and are useful to identify spatial imbalances. For example, Figures 18(a) and 18(b) present density maps for LU.D on 1024 cores. In Figure 18(a), it can be clearly seen that the number of MPI_Send calls issued by the benchmark is correlated with the number of neighbours in the mesh (see Figure 17(e) for the associated topology). Dealing with the total size, Figure 18(b) shows a small imbalance which seems to follow the LU decomposition pattern. Figures 18(c), 18(d) and 18(e) present density maps for BT.D on 8281 cores. This example shows an imbalance in total MPI point-to-point size (see Figure 18(e) with blue at 660.93 MB and red at 664.87 MB). Interestingly, times spent in MPI wait calls (Figure 18(d)) and in collectives (Figure 18(c)) follow the same symmetry with nearly twice as much time between red (491.8 ms) and green areas (288.5 ms) — suggesting a possible computational imbalance. Although empirical, observations made upon Figure 18 can provide

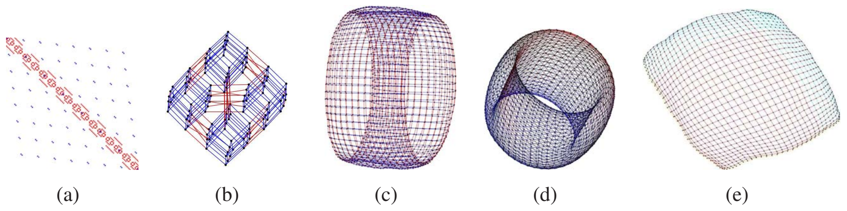
Figure 17. Sample outputs from the topological module as generated by our analyser, invoking the Graphviz [24] tool for Figures b to e.

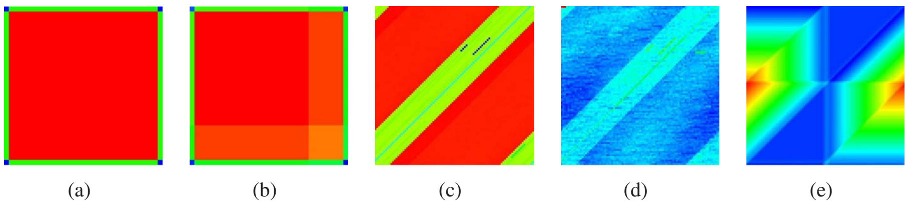
Figure 18. Sample outputs from the density map module as generated by our analyser.

spatial insights on codes, helping developers to understand and observe the consequences of their balancing policies.

We acknowledge that all those analysis could have been done locally with probably less overhead instead of moving a large amount of data over the network. But, as the purpose of this work is to build basic blocks for a centralised *continuous profiling* engine, we deliberately avoided to constrain the analysis too early in the measurement chain. For example, we are working on a wait-state analysis which will take advantage of a distributed blackboard — these simple analysis shall therefore be seen as a preliminary implementation of this concept.

V. Related Work

Reducing instrumentation overhead and trace size is a concern common to any performance tool, leading to a wide variety of approaches competing to find the best trade-off between collected data verbosity and instrumentation overhead. Collecting a lot of data allows careful post-mortem exploration of program's behaviour, as in tools like Vampir [23] or the Intel Trace Analyser and Collector [25] which are able to display the actual temporal behaviour of an application, at the cost of large storage requirements. On the other hand, some tools adopted a purely on-line approach. Validation tools such as Marmot [6] or MUST [5] can perform on-line MPI usage validation through the PMPI interface. An example of purely online profiling tool is mpiP [7] which relies on a statistical communication aggregate which is reduced at the end of execution in order to generate an MPI interface profile. Naturally, embedded instrumentation cannot perform complex analysis without impacting the target program. Therefore performance tools such as Scalasca [8], [21] or Tau [9], [26] combine several online reduction techniques with post-mortem analysis.

Blackboard system has been used for profiling purposes in extension to HPCToolkit [27], coupling multiple tools at node level through a shared memory segment. This process allows the generation of *augmented events* which are processed in a topology-aware manner. Our approach differs in that the blackboard is not located at node level but in a set of processes which can be indifferently accessed, allowing multi-analysis, while analysis costs which would otherwise impact local processes are offloaded.

Dealing with the online approach, several tools rely on a Tree Based Overlay Network (or TBON) which consists of a data reduction through filters at each level of the tree. In this case, instrumented processes are part of leaf nodes (called back-end) and data are streamed to the front-end (root node) while being processed by several reduction filters. This method is used in the DDT debugger [28] for control and program state reduction purposes. Frameworks such as Paradyn MrNET [29] can be used to build efficient TBONS with arbitrary reduction filters. This method proved its scalability in the STAT debugger [30]. TBONs are also used in the Generic Tool Infrastructure (GTI) [31] which relies on P^NMPI and provide a generic infrastructure to instrument and reduce events. The GTI allows efficient generation of event instrumentation, transport and reduction through XML specifications. It has been successively used to offload and parallelize MUST's validations. Examples of profiling tools using the TBON paradigm are Periscope [32] and MAP (derived from DDT) which both operate a tree-based reduction on performance metrics. Another tool which performs online analysis is Paradyn [33]. It relies on a multi-threaded "Performance Consultant" in order to instrument and analyse application events thanks to a network-based coupling which is also implemented over MrNet. To our knowledge it relies on TCP sockets (MrNet 4.0.0). Our approach is similar, although we take advantage of high performance networks (thanks to MPI) and rely on a distributed data-flow engine. Consequently, although TBONs can perform efficient and scalable [30] distributed reductions, we focus on maximising the bisection bandwidth between partitions. In this purpose, applications are mapped to all analysis processes (not only the back-end as in TBONs) and we provide a parallel data-flow engine to process data. Moreover, as the analysis partition has a fully functional MPI, the blackboard approach does not limit data processing to tree based reductions, and covers a wider usage spectrum, ranging from simple event notification to distributed data-flows, including TBONs.

As far as profiling tools based on fine-grained events are concerned, they generally either consider the file-system as a compulsory intermediate between instrumentation and analysis or process measurement data locally. A lot of work has been done to reduce instrumentation overhead [8]–[10], [21], [26], [34]–[36] and efficiently store these data in various formats, relying on parallel IO libraries [2]. Our work differs in the sense that overcoming IO limitations is achieved by replacing them with a more effective coupling mechanism. In comparison with existing trace formats [22], our event representation structure is very simple as the C structure is directly sent. Nonetheless, this simple strategy manages to scale up to several thousand cores, while providing analysis similar to post-mortem ones in a reduced time-frame, thanks to pipe-lining. Streamed analysis is very close to post-mortem analysis as it is decoupled from the execution and can be distributed. Still, we are aware that our methods does not offer the ability to replay traces for exploration or iterative research and does not provide global informations such as application wall-time or total number of threads. In comparison with other approaches, this work is then an attempt to replace the classical post-mortem performance tool work-flow by a much simpler method which provides wider parallelism.

VI. Conclusion and Future Work

This paper presents an online trace reduction framework which relies on online coupling between instrumentation and analysis. As this method does not depend on tracing, it completely avoids the IO bottleneck and allows pipe-lining of performance analysis, reducing its overall cost.

To achieve this goal, a generic online coupling mechanism relying on MPI in MPMD mode has been exposed in sections II-A and III-A. It provides MPI virtualization, application mappings and communications components to our online analysis framework. Dealing with performance measurements analysis, a data-centric tasks engine inspired by blackboard systems (section II-B and III-B) is in charge of performing analyses in parallel. Thanks to our multi-level blackboard architecture, our distributed analysis engine is able to concurrently profile multiple applications. In Section IV-B, we demonstrate the scalability of this method up to 4096 cores on both Tera 100 and Curie supercomputers. Then, Section IV-D, shows sample outputs from online analysis modules. These outputs are gathered in a report structured with one chapter per instrumented application.

By proving the feasibility of a runtime coupled profiling server, this first implementation sets the basis for a truly machine wide server which could provide profiling as a service. Our future works include the investigation of similar runtime coupling at the communication layer level. Dealing with the analyser, we are already working on the implementation of a module, acting as an IO proxy, to generate selective traces in the OTF2 format in order to combine our analysis with existing tools such as Vampir Trace. We are also extending our Blackboard implementation to support distributed analysis, extending data-flow outside of nodes boundaries thanks to a one-sided communication scheme. It opens opportunities for distributed analysis algorithms including stateful analysis. Other modules are also studied, with a particular interest for those performing "real-time" performance analysis.

Acknowledgment

This work is (integrated and) supported by the PERFCLOUD project. A French FSN (Fond pour la Société Numérique) cooperative project that associates academics and industrials partners in order to design then provide building blocks for a new generation of HPC datacenters.

References

[1] MPI-Forum, "MPI: A message passing interface standard, version 2.1," 2008.

[2] W. Frings, F. Wolf, and V. Petkov, "Scalable massively parallel I/O to task-local files," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, 2009.

[3] "NetCDF : Network Common Data Form." [Online]. Available: http://www.unidata.ucar.edu/software/netcdf/

[4] The HDF Group. (2000-2010) Hierarchical data format version 5. [Online]. Available: http://www.hdfgroup.org/HDF5

[5] "MUST: A Scalable Approach to Runtime Error Detection in MPI Programs," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds., 2010.

[6] B. Krammer, M. S. Müller, and M. M. Resch, "MPI Application Development Using the Analysis Tool MARMOT," in *In ICCS 2004, volume LNCS 3038*. Springer, 2004.

[7] J. S. Vetter and M. O. McCracken, "Statistical scalability analysis of communication operations in distributed applications," in *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, 2001.

[8] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurr. Comput. : Pract. Exper.*, 2010.

[9] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *Int. J. High Perform. Comput. Appl.*, 2006.

[10] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski, "ScalaTrace: Scalable compression and replay of communication traces for high-performance computing," *J. Parallel Distrib. Comput.*, vol. 69, no. 8, Aug. 2009.

[11] A. Calotoiu, C. Siebert, and F. Wolf, "Pattern-Independent Detection of Manual Collectives in MPI Programs," in *Proc. of the 18th Euro-Par Conference, Rhodes Island, Greece*, ser. Lecture Notes in Computer Science, vol. 7484. Springer, Aug. 2012, pp. 28–39.

[12] R. Engelmore and T. Morgan, *Blackboard systems*, ser. Insight series in artificial intelligence. Addison-Wesley, 1988.

[13] D. D. Corkill, "Collaborating Software: Blackboard and Multi-Agent Systems & the Future," in *Proceedings of the International Lisp Conference*, 2003.

[14] ——, "Design Alternatives for Parallel and Distributed Blackboard Systems," Amherst, MA, USA, Tech. Rep., 1988.

[15] L. D. Erman and V. R. Lesser, "The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty," *Computing Surveys*, vol. 12, pp. 213–253, 1980.

[16] H. P. Nii, E. A. Feigenbaum, J. J. Anton, and A. J. Rockmore, "Readings from the AI magazine," R. Engelmore, Ed., 1988, ch. Signal-to-symbol transformation: HASP/SIAP case study.

[17] TOP 500. (2010) Tera 100 Supercomputer. [Online]. Available: http://top500.org/system/10589

[18] ——. (2012) Curie Supercomputer (thin nodes). [Online]. Available: http://top500.org/system/177818

[19] M. Schulz and B. R. de Supinski, "PNMPI tools: a whole lot greater than the sum of their parts," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[20] M. Wolff, S. Jaouen, and L.-M. Imbert-Gérard, "Conservative numerical methods for a two-temperature resistive MHD model with self-generated magnetic field term," in *CEMRACS'10 research achievements: Numerical modeling of fusion*, 2011.

[21] Z. Szebenyi, T. Gamblin, M. Schulz, B. R. de Supinski, F. Wolf, and B. J. N. Wylie, "Reconciling Sampling and Direct Instrumentation for Unintrusive Call-Path Profiling of MPI Programs," in *IPDPS, Anchorage, AK, USA*. IEEE Computer Society, 2011.

[22] D. an Mey, S. Biersdorff, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S. S. Shende, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A Unified Performance Measurement System for Petascale Applications," in *Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany, June 2010*, 2012.

[23] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR : Visualization and Analysis of MPI Resources," *Supercomputer*, 1996.

[24] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software Practice and Experience*, vol. 30, no. 11, 2000.

[25] "Intel Trace Analyzer and Collector." [Online]. Available: http://software.intel.com/en-us/intel-trace-analyzer

[26] A. Morris, A. D. Malony, S. Shende, and K. Huck, "Design and Implementation of a Hybrid Parallel Performance Measurement System," in *Proceedings of the 2010 39th International Conference on Parallel Processing*, 2010.

[27] A. Mandal, R. Fowler, and A. Porterfield, "System-wide Introspection for Accurate Attribution of Performance Bottlenecks," in *Proceedings of the Second International Workshop on High-performance Infrastructure for Scalable Tools*, ser. WHIST 2012.

[28] "Allinea Distributed Debugging Tool." [Online]. Available: http://www.allinea.com/products/ddt/

[29] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, ser. SC '03. ACM, 2003.

[30] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz, "Scalable temporal order analysis for large scale debugging," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 44:1–44:11.

[31] T. Hilbrich, M. S. Müller, B. R. de Supinski, M. Schulz, and W. E. Nagel, "GTI: A Generic Tools Infrastructure for Event-Based Tools in Parallel Systems," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, ser. IPDPS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1364–1375.

[32] S. Benedict, V. Petkov, and M. Gerndt, "PERISCOPE: An Online-Based Distributed Performance Analysis Tool," in *Tools for High Performance Computing 2009*, M. S. Mller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Springer Berlin Heidelberg, 2010, pp. 1–16.

[33] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The paradyn parallel performance measurement tool," *Computer*, vol. 28, no. 11, 1995.

[34] N. R. Tallent, J. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto, "Scalable fine-grained call path tracing," in *Proceedings of the international conference on Supercomputing*, 2011.

[35] A. Knupfer, "Construction and Compression of Complete Call Graphs for Post-Mortem Program Trace Analysis," in *Proceedings of the 2005 International Conference on Parallel Processing*, 2005.

[36] J. Gonzalez, J. Gimenez, and J. Labarta, "Automatic detection of parallel applications computation phases," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–11.