Effective Performance Measurement at Petascale Using IPM

Karl Fürlinger
University of California at Berkeley
EECS Department, Computer Science Division
Berkeley, California 94720, USA
Email: fuerling@eecs.berkeley.edu

Nicholas J. Wright and David Skinner
NERSC
Lawrence Berkeley National Laboratory
Berkeley, California 94720, USA
Email: {njwright, deskinner}@lbl.gov

***Abstract*—As supercomputers are being built from an ever increasing number of processing elements, the effort required to achieve a substantial fraction of the system peak performance is continuously growing. Tools are needed that give developers and computing center staff holistic indicators about the resource consumption of applications and potential performance pitfalls at scale. To use the full potential of a supercomputer today, applications must incorporate multilevel parallelism (threading and message passing) and carefully orchestrate file I/O. As a consequence, performance tools must also be able to monitor these system components in an integrated way and at the full machine scales.**

We present IPM, a modularized monitoring approach for MPI, OpenMP, file I/O, and other event sources. We describe its implementation design principles, which are targeted for efficiency and minimal application perturbation, and present an application study of using IPM at scale.

I. Introduction

Moore's Law continues unabated; the number of transistors on a chip still doubles every eighteen months. However, because of power constraints, clock speeds remain approximately constant and the extra transistors are being used to add cores. Therefore supercomputing systems are being built with an ever increasing number of processing elements. Doubling roughly every thirteen months, the performance increase of supercomputers has outpaced the predictions of Moore's law, primarily due to increased level of parallelism [1]. This trend will only be exacerbated by the widespread adoption of multi- and manycore CPUs in the future. In fact, exascale systems are predicted to feature millions of compute cores. It is therefore clear that some form of multilevel parallelism will play a major role in the programming of these systems. A second trend is decreased overall I/O capability. The ratio of I/O bandwidth to FLOPS is decreasing and in an exascale timeframe are expected to be at least an order of magnitude worse, relatively speaking, than today. Therefore I/O performance is becoming an increasingly important factor. To understand all these potential complex performance issues and their interactions there is a clear need for tools that will allow application developers to gain an understanding of their performance issues.

Tools for detecting performance issues in High Performance Computing (HPC) environments are available in a wide variety of forms, purposes and scopes of application. Which tool to use depends largely on what sort of problem one is addressing. To use an analogy from motor vehicles: There are complex tools used by mechanics and simple diagnostic tools (eg. warning lights) used by their regular operators. We contend that there is similarity to the motor vehicle case and room for both modes of operation in the HPC tools arena. This view has been motivated by many years of experience at HPC centers that teaches us that if obtaining performance information is onerous, if it involves an appointment with a mechanic, it is much less likely to happen. In previous work we have described our solution to this problem - the Integrated Performance Monitoring (IPM) framework [2], [3]. To date, primarily because of our focus on ease of use, this has been used by more than 310K batch job performance profiles have been collected on NERSC machines over the past 6 years. This focus on ease of use and providing a compact overall view of performance has also led to IPM being used in several commercial HPC vendors to allow them to gain an understanding of their potential customers codes within the context of procurements.

The collected IPM performance profiles provide a rich data set for exploring topics of interest of managers of supercomputing centers. By analyzing the data they contain about memory, compute and network usage it is possible to make vital decisions about where to provision more resources to remove bottlenecks, as well as allowing the determination of attractive features for future machine procurements. For example, with the current pressures on memory per core, an understanding of an HPC center's workloads memory requirement is crucial.

In this paper we present a re-architected implementation of our existing workload and performance analysis tool IPM which features a modularized design and adds monitoring modules for OpenMP, and file I/O operations. IPM retains the focus on ease of use and avoiding application perturbation while focusing less on a drill-down into the applications than other tools.

The main contributions of this paper are:

- We present an integrated framework for the simultaneous analysis of the three most important aspects of performance on today's supercomputers (communication, threading, file I/O).
- We describe our tool IPM that implements this framework

in a modularized and extendable architecture.
- We show that we achieve highly efficient monitoring with very low overheads and perturbation of the target application.
- We demonstrate that our tool is scalable to the full size of contemporary supercomputers.

The rest of this paper is organized as follows: In Sect. II we introduce the design and implementation of IPM. We describe the event data sources, the event processing and storage, and the data processing and analysis. In Sect. III we evaluate IPM at scale on a nuclear fusion simulation code. Due to space restrictions we focus on the OpenMP and MPI aspects in this paper. We describe related work in Sect. IV and conclude in Sect. V with an outlook on future work.

II. Performance Monitoring with IPM

A. IPM Design Principles

The objective of IPM is to deliver an inventory of program execution events in sufficient detail to inform the user about performance while introducing minimal application overheads. The general model we assume is that of an application comprised of n potentially multi-threaded processes with events of interest happening in these processes. Potential sources of events are the sending and receiving of messages using MPI, collective operations, file I/O operations or the execution of OpenMP-parallel regions.

While it can be configured to write traces to a log file as well, IPM's strength lies in its profiling mode where the time stamps of individual events are not of interest and only statistics of event durations are collected. To efficiently process and store the potentially very large number of events occurring in an application, we derive a unified event signature and encode it as a bit vector. The event signature contains the critical information about the event we are interested in. Fig. 1 shows the structure of an event signature bit vector as used by IPM with 128 bits.

- **Event ID:** Corresponds to the numeric encoding of the event being monitored (completion of an `MPI_Send` command, execution of an OpenMP parallel region, etc.). 12 reserved bits correspond to 4096 different types of supported events.
- **Region ID:** Users can manually mark regions of interest using the `MPI_Pcontrol` mechanism. IPM implements these calls and creates an internal data structure to represent the region. Event statistics are then computed globally (for the entire application) as well as for each marked region separately.
- **Thread ID:** This 8 bit field encodes the ID of the thread in which the monitored event originated. In IPM there is currently only one hash table per process, which is manipulated exclusively in the sequential portions of the application. Multithreaded data is kept in thread-safe data structures local to a module (cf. Sect. II-C) until a sequential region is reached, at which point the central hash table is updated.
- **Callsite ID:** IPM tries to derive the call site of a monitored function by walking the call stack using `libunwind` or other stack walking libraries. Recording the call site has the benefit of being able to differentiate between MPI calls with the same signature (communication partner, message size, ...) but with different dynamic contexts.
- **Partner ID:** This field encodes the communication partner for MPI operations and the file ID for file operations. For collective MPI operations this entry holds the root of the operation and for point to point operations the peer rank is either derived from the arguments to the MPI call or by examining the `MPI_Status` structure.
- **Buffer/Message Size:** This field encodes the length of the message for MPI operations and the number of bytes read or written for I/O operations. 32 bits are reserved for this field, corresponding to buffer sizes of up to four gigabytes per operation.

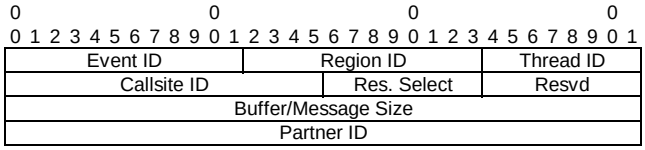


Fig. 1: The structure of the 128 bit event signatures.

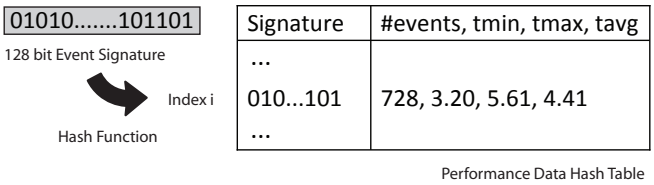


Fig. 2: Event signatures and the hash table used by IPM.

IPM observes events at runtime as they are happening in the application, computes their signature and updates the event's statistics in a performance data table. In general, many events may be mapped to the same signature, such as an MPI message exchange between communication partners in a loop with the same parameters. To store and process the performance data efficiently, IPM uses a hash table to implement the performance data table. The event signature is used as the hash key and the hash values are the number of occurrences, the minimum, maximum, and sum of the duration of the events (cf. Fig. 2). Once events from different sources are stored in the hash table, they can be processed and analyzed in a uniform manner.

B. Performance Data Event Sources

To monitor MPI events, we use the standard PMPI interface. Currently we monitor file I/O at the level of the standard C library calls (`fopen()`, `fclose()`,...) and Unix system calls (`open()`, `close()`,...), by using dynamic library interposition [4] or wrapping these calls at link time for static binaries.

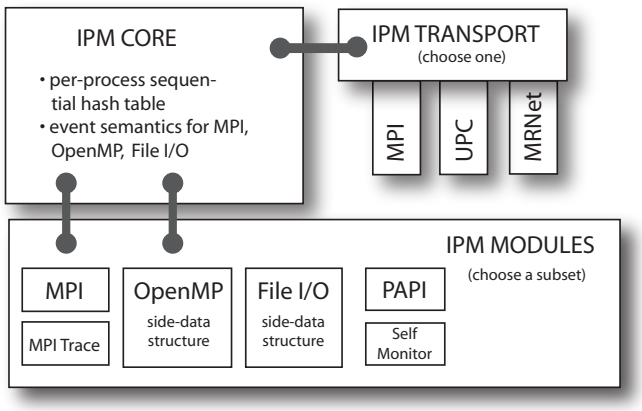


Fig. 3: The modularized design of IPM.

To monitor OpenMP regions we rely on compiler inserted instrumentation as provided by the PGI and Cray compiler suites. If instructed, the compiler will insert instrumentation points in and around OpenMP constructs that are implemented and monitored by IPM. The structure of these calls is largely similar to the POMP calls [5] added by the OPARI source-to-source instrumenter [6]. As an important difference, the source code instrumentation approach requires a recompilation and often reaches its limits when users perform non-standard preprocessing on their source code, such as the inclusion of OpenMP pragmas from header files. The compiler-based instrumentation employed by IPM has no such limitations.

C. Module Concept

In order to adapt to different requirements and system capabilities, IPM is designed as a modularized system. The schematic system architecture of IPM is shown in Fig 3. The IPM core module is responsible for initialization of centralized data structures such as the hash table and for registering all available modules. IPM modules are a mechanism of compile-time specialization and not dynamically loadable components, although some runtime configuration is possible using environment variables.

The transport module refers to the mechanism used by IPM to aggregate results across tasks. Since IPM can be used to monitor a parallel application on a per-process level, performance data is co-located with the application processes and needs to be aggregated to compute the IPM application report and log file. Currently the transport module is available for MPI only, but nothing precludes the usage of other communication mechanisms, such as UPC or MRNet [7].

The monitoring components for MPI, OpenMP, and file I/O are each implemented as a separate module. Any subset of these feature modules can be selected to form a valid IPM installation. Notably, the monitoring of file I/O is available for purely sequential jobs without any remaining dependency on MPI (i.e., such an IPM installation can be built on machines without having an MPI compiler and runtime installed). We have used this feature successfully in collaboration with LBNL and CERN scientists analyzing the AthenaMP code [8] which is used for processing data from the ATLAS particle physics experiment.

A number of other modules are currently under construction, such as an MPI-IO component, a module for monitoring the CUDA runtime for GPU enabled systems, and a module for analyzing network interface counters such as those found on Infiniband hardware. Regular (CPU) hardware performance counters are accessed using a module that encapsulates PAPI [9] functionality. IPM supports component PAPI and reads an environment variable `IPM_HPM` for a user-specified list of counters. Any combination of counter names (from several PAPI components, such as `PAPI_FP_OPS` for floating point operations of and `ETH0_RX_BYTES` for the number of received bytes on the first Ethernet adapter) can be specified and IPM will keep track of which event maps to which PAPI component.

Finally, there is module for self-monitoring of IPM's activity. If enabled, an appendix to IPM's profiling report is available detailing some of the important internal runtime statistics such as hash table fill rates, time in `MPI_Init` and `MPI_Finalize` on IPM's behalf and communication amount and time used for performance data aggregation by IPM.

D. Performance Data Output and Post-Processing

IPM's output comes in two forms, each of which can be configured to be disabled, or delivered in a `full` or `terse` format. The most basic output IPM delivers is a banner written to the terminal immediately upon application exit which holds some of the most important high-level job metrics. These include the consumed wallclock time, the number of processes, threads, and nodes the job ran with, and the overall percentage of time spent in MPI, OpenMP parallel regions, and for performing file I/O. Straight forward access to high level metrics is an important aspect of IPM's philosophy as it allows users and computing center staff to get an idea of the overall execution characteristics and resource consumption of an application.

By setting the environment variable `IPM_REPORT` to `full`, a more detailed version of this banner is printed, containing individual events ranked by their contribution. An example showing the full banner of an application is given in Fig. 4. The full banner details the contributions to the wallclock time by MPI and file I/O calls and the time spent in parallel regions as the minimum, maximum, and average over all ranks. The distribution of the time in OpenMP (`OMP`) and the idle time in OpenMP parallel regions (`OMP idle`) at the end of parallel and worksharing regions is displayed in a similar way. The last section lists all individual contributing events sorted by their summed wallclock time. The special event `OMP_PARALLEL` refers to the execution of a parallel region and the `[count]` column refers to the number of executions of this parallel region.

Analyzing the performance of individual processes is possible by requesting an IPM log file. This file in XML format holds detailed information for about each rank's events and

```
##IPM##################################################
#
# command   : ./a.out
# start     : Sun Mar 14 16:55:39 2010  host      : nid01829
# stop      : Sun Mar 14 17:04:33 2010  wallclock :   533.12
# mpi_tasks : 2048 on 1024 nodes        %comm     :    29.41
# omp_thrds : 6                         %omp      :    50.63
# files     : 12                        %i/o      :    12.09
# mem [GB]  : 2774.44                   gflop/sec :   418.58
#
#           :      [total]       <avg>         min         max
# wallclock :   1091671.57      533.04      532.99      533.12
# MPI       :    321034.43      156.76      109.03      239.23
# I/O       :    131947.08       64.43       11.83      113.87
# OMP       :    552665.28      269.86      205.07      305.36
# OMP idle  :     48262.98       23.57       21.30       27.40
# %wall     :
#   MPI     :                     29.41       20.45       44.88
#   OMP     :                     50.63       38.47       57.28
#   I/O     :                     12.09        2.22       21.36
# #calls    :
#   MPI     :     76235998       37224       37223       37320
# mem [GB]  :      2774.44        1.35        1.35        1.36
#
#                          [time]       [count]       <%wall>
# OMP_PARALLEL         552665.28     131439989         50.63
# MPI_Allreduce        247648.04      14438400         22.69
# fread                 69813.27       5488640          6.40
# ...
#
##########################################################
```

Fig. 4: An example full application banner as delivered by IPM.

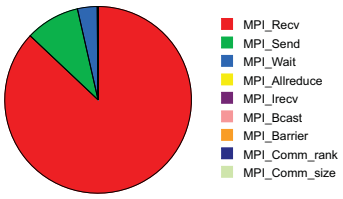
includes a full copy of the hash table if `IPM_LOG=full` is specified. The XML file is written sequentially by a designated application process (rank 0 by default) which receives each process' performance information, one by one. At high concurrencies a parallel writing scheme is employed. In this case MPI-IO is used and each rank writes to its portion of the log file in parallel. This mechanism is very efficient and scales well to the full size of machines. For example, we have observed that writing the full IPM log at 72 000 processes takes less than 2 minutes on the Cray XT5 'Kraken' at Oak Ridge National Laboratory.

E. HTML Profiling Report

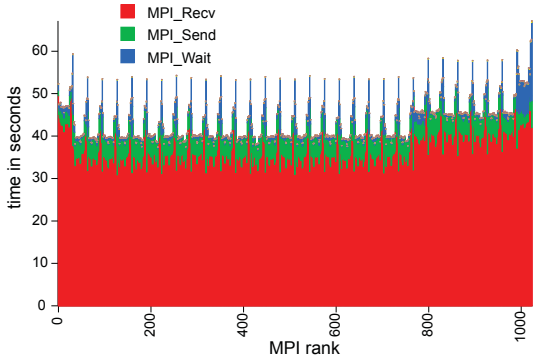
The IPM XML file is input to a parser that generates an HTML representation of the profiling report. Using standard HTML to visualize performance data has the advantage that no special graphical user interface (GUI) is required to view the data. A new parser and techniques using advanced HTML/Javascript charting to address the scalable visualization of data at high concurrencies are currently under development.

Among others, the HTML profiling page contains these entries:

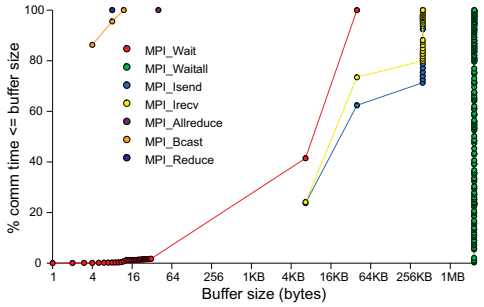
- The data contained in the text-based banner is reproduced in a table at the top of the profiling report.
- A pie chart (Fig. 5a) displays the breakdown of the total MPI time into the various contributing MPI calls such as `MPI_Allreduce` or `MPI_Wait`.
- For each monitored hardware counter event, the minimum, maximum and average values are displayed as well as the location (rank) of where the minimum and maximum values are achieved.



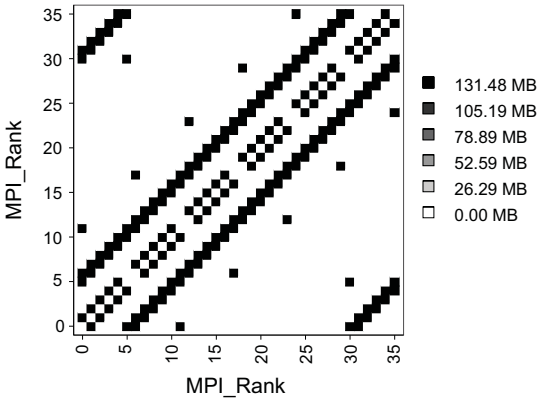
(a) MPI pie chart.



(b) Time in MPI routines by rank.



(c) Cumulative distribution graph of buffer sizes.



(d) Communication topology graph.

Fig. 5: Some of the performance data displays provided by IPM.

- A load balance line graph showing the consumed DRAM memory, floating point rate, and wallclock time. The horizontal axis is the rank dimension and the graphs are available both in sorted (by memory, FLOPS, wallclock time), as well as unsorted (natural rank order) variant.
- A stacked load balance graph shows the breakdown of the MPI time into individual MPI calls over the rank dimension. An example for this graph (sorted by rank) is shown in Fig. 5b. This type of display is especially useful to identify and locate load imbalance situations.
- Cumulative distribution graphs provide an understanding of the message size distribution of an application. The horizontal axis is the buffer size n used in the operation and the vertical axis denotes how many calls have had a buffer size smaller or equal to n.
- A similar cumulative distribution graph is also provided where the accumulation is not performed over the number of calls but the time spent in the messaging operation instead. The graph in Fig. 5c shows an example.
- A communication topology graph as shown in Fig. 5d. This graph shows the amount of data exchanged between a pair of processes. The sending process is depicted on the horizontal axis, the receiving process is shown on the vertical axis.

III. Evaluation

In order to illustrate the capabilities and performance of IPM we first present a study of the efficiency and overheads of IPM and then present an application case study using GTC (a code for the simulation of nuclear fusion) on three different architectures with different software environments.

A. *Efficiency of IPM*

We evaluate the efficiency of IPM using three different experiments. First we evaluate the raw hash table performance and then try to answer the question of how many events IPM can process per second taking hash table operations and event timing into account. Finally, we study the effect IPM monitoring has on an application by conducting an ensemble study of the MAESTRO application with and without IPM enabled. We find that on average IPM causes an overhead of less than 0.25% for the application kernel of MAESTRO and less than 0.5% for the whole application (including writing the profiling report). On most HPC resources today these perturbations are indistinguishable from natural runtime variations observed due to contention for shared resources.

Fig. 6 shows the raw hash table performance of IPM on an Intel Nehalem based system. 10 Million events are prepared with a varying number of unique keys (corresponding to the horizontal axis) and stored in the hash table with 32K entries, including updating the counts and timing statistics, but not actually getting time stamps. The performance depends heavily on the compiler and the optimization flag used but with reasonable settings for production code IPM can process the 10 Mio events in less than 0.8 Seconds. Performance depends on the number of unique keys used, as collisions become more likely when the hash table fill rate approaches 100%. In the vast number of application test cases we have observed with a 32K hash table, the fill rate is well below 50%.

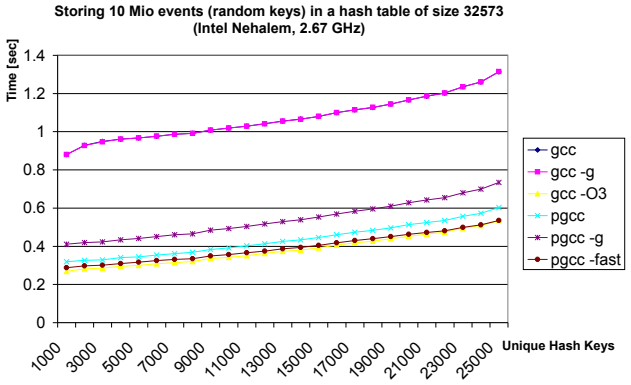


Fig. 6: Raw hash table performance of IPM.

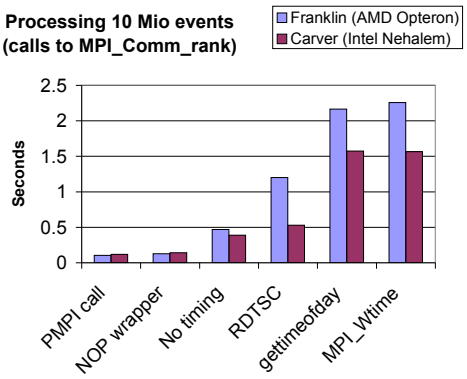


Fig. 7: The influence of timing routines on the achievable event rate.

Fig. 7 highlights the contribution of acquiring time stamps as part of the monitoring process. Here we again record 10 Million events, this time by calling `MPI_Comm_rank()` in a simple MPI program. The bars in the graph show the timings on two different systems at NERSC, one based on an AMD Opteron processor, the other one based on Intel Nehalem. The leftmost pair of bars forms the baseline. In this case, the application actually calls `PMPI_Comm_rank()` to bypass the monitoring altogether. The bars to the right correspond to a NOP wrapper i.e., the call is wrapped by IPM but no monitoring is performed and the routine returns immediately. In the next experiment all monitoring actions are performed but time stamps are not acquired. Finally, the last three bars show the influence of three different sources for timing data. `RDTSC` corresponds to the low level time stamp counter on modern microprocessors, `gettimeofday` is an operating system timing routine and `MPI_Wtime` is MPI's timing mechanism. Evidently the last two experiments show comparable results on both systems, suggesting that

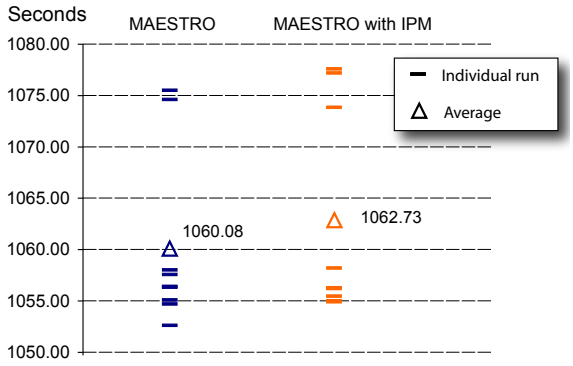


Fig. 8: Ensemble study using the MAESTRO application.

`MPI_Wtime` is in fact based on `gettimeofday`. `RDTSC` is the more efficient timing mechanism on both platforms.

To summarize: as a ballpark number we claim that IPM can handle at least roughly 10 Million events per second per application process on a contemporary platform. This number should exceed the messaging rate of an individual process in today's HPC applications.

In the last efficiency experiment we tested the overall influence that IPM has on the runtime of an application. Previous experiments indicated that MAESTRO from the NERSC-6 benchmark suite was the most demanding application and so we ran MAESTRO with 256 MPI processes 9 times with and 9 times without IPM monitoring. Fig. 8 shows the results. On the left hand side are the runs without IPM and on the right side are the runs with IPM. The vertical axis is the runtime of the application kernel (not including startup and shutdown), a short line is plotted for each individual run. We see that the average runtime increased from 1060.08 to 1062.73 seconds (less than 0.25%) and that this is well below the natural variation in runtimes for the same binary on this system. For the full application (this includes writing the log file to disk) the average increase in runtime is less than 0.5%.

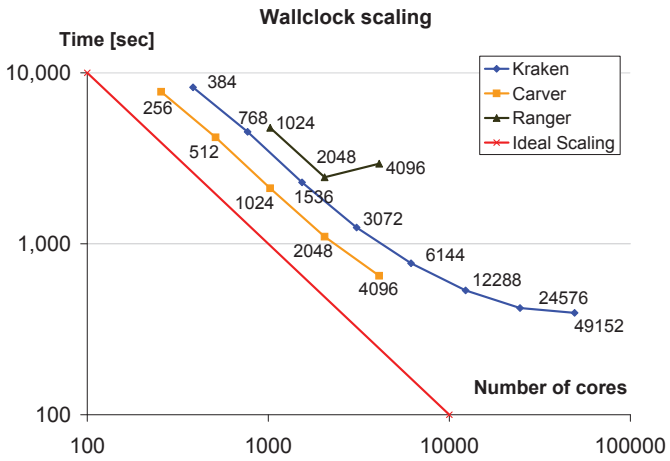
B. Application Study: GTC

The Gyrokinetic Torodial Code (GTC) is used to perform 3D Particle In Cell (PIC) simulations of gyrokinetic plasma microturbulence [10]. In this case study we use the NERSC6 GTC benchmark code. The benchmark problem simulates 16×10^6 ions and electrons on a grid of size $1.35\text{x}10^6$ and runs for 50 timesteps.

We ran GTC on three different machines:

Kraken is a Cray XT5 machine based at Oak Ridge National Laboratory. It has hex-core AMD Istanbul processors running at 2.3GHz, with dual socket nodes and a Cray Seastar 2+ interconnect.

Ranger is a Sun constellation cluster based at the Texas Advanced Computing Center. It has quad-core AMD Barcelona processors running at 2.3 GHz, with quad-socket nodes and a single Single Data Rate (SBR) Infiniband link per node to a Sun Constellation IB switch that has a full Clos fat-tree topology.



(a) Wallclock scaling of GTC.

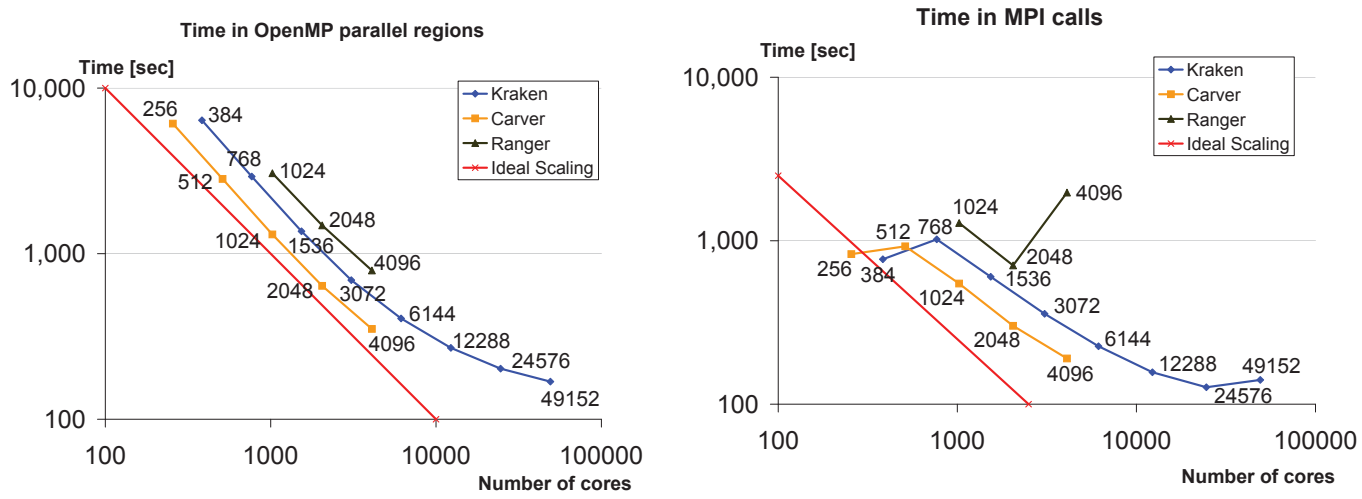
Fig. 9: Overall scaling of GTC on the three machines.

Carver is an IBM iDataPlex cluster based at NERSC. It has quad-core Intel Nehalem processors running at 2.67 GHz, with dual socket nodes and a single Quad Data Rate (QDR) IB link per node to a network that is locally a fat-tree with a global 2D-mesh.

Fig. 9 shows the overall strong scaling plot of GTC on the three machines, using four OpenMP threads per MPI task on Carver and Ranger and six on Kraken. Carver is the fastest, followed by Kraken and then Ranger. The scaling plot also shows that on Kraken the scalability of this benchmark tails off above about 12K cores as the individual tasks run out of parallel work, Carver seems to follow the same trend (due to the smaller size of the machine we were not able to collect numbers above 4K), while there is already a considerable *slowdown* when going from 2K to 4K processors on Ranger.

Using IPM we were able to measure the performance of GTC on all three platforms up to a maximum concurrency of 49,152 cores. The overall performance breakdown obtained using IPM is shown in Fig. 10. Fig. 10a shows the scaling of the OpenMP portion of the runtime and Fig. 10b shows the scaling of the MPI time. These give some indication as to reasons that Carver is the fastest machine overall, it has both the fastest processors and the fastest interconnect, whereas Ranger has both the slowest processors and interconnect. The OpenMP part of the GTC code scales very well on all three machines up until the code starts running out of parallel work. In contrast, Ranger gets hit badly by the slow interconnect at 4096 cores and this is the primary reason why the code does not scale up to 4K cores on Ranger (the compute part scales well, the communication part does not).

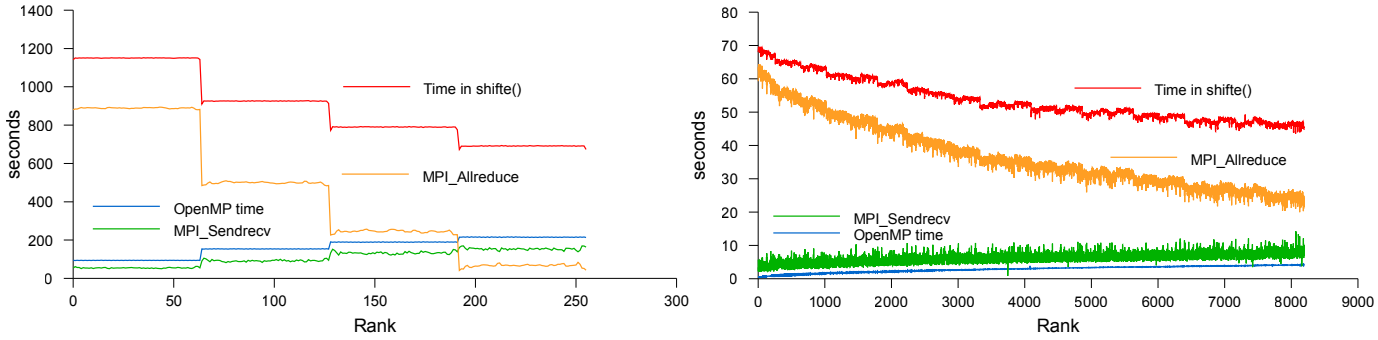
The internal timing routines within GTC indicate that a majority of the time for this benchmark in two routines, `shifte()` and `pushe()`. Using the region functionality of IPM described in Section II we added two additional lines of code to GTC to allow us to obtain separate IPM profiles for the `shifte()` routine. Some of this profiling data is shown in Fig. 11. This shows that the `shifte()` routine



(a) Scaling of time in OpenMP.

(b) Scaling of MPI time.

Fig. 10: Scaling behavior of GTC with respect to overall wallclock time, time in MPI routines, and in OpenMP-parallel regions for three different machines.



(a) 256 Processes/6 Threads.

(b) 8192 Processes/6 Threads

Fig. 11: Load balance of GTC's `shifte()` routine at 1536 cores and 49152 cores.

is substantially load-imbalanced across the processes, that the imbalance is related to the number of particle domains in the code, and that the imbalance is caused by the loop(s) immediately preceding the MPI_Allreduce. Upon further investigation this did indeed turnout to be the case.

Based on this result we developed an alternative particle distribution layout for GTC. Using this alternative distribution indeed resulted in much improved load balance and reduced time in the `MPI_Allreduce` collective operation substantially. However, due to reduced spatial locality of this new particle distribution, the sequential performance suffered from an increased number of cache misses, erasing the gains in communication performance. Improving load balance while retaining cache locality in GTC is ongoing work.

IV. Related Work

There are a number of related performance analysis tools for scientific MPI and OpenMP applications. TAU [11], [12] is a versatile tool set for MPI, OpenMP and Pthreads codes using source instrumentation for function level data collection. HPCToolkit [13], [14] is a tool based on binary analysis and statistical sampling which has its strengths in the detection of code hotspots but places less focus on analysis of communication behavior. Scalasca [15], [16] is a tool for tracing of MPI and OpenMP applications based on a replay of the communication events and the identification of common patterns of inefficiency. Runtime summarization (profiling) capability was added recently. In addition to tools from academic and research labs, vendor-provided tools such as Cray PAT [17] and Intel Trace Collector, Trace Analyzer, and Thread Profiler [18], [19] are available but frequently limited to their particular platforms. ompP [20] is full-featured OpenMP profiling tool that relies on source code instrumentation using the Opari [6] source-to-source instrumenter. In the case of IPM the limitations and shortcomings of source-to-source instrumentation are not acceptable, and as a consequence IPM uses compiler inserted instrumentation to gather high level OpenMP metrics.

In contrast to the tools mentioned above, IPM is designed to give a high level overview of application activity while introducing minimal overheads and being usable in a "flip of the switch" fashion. Given a compatible setup, enabling IPM can be accomplished with a single command such as `module load IPM` . This usage scenario precludes the usage of source code instrumentation (TAU), trace replay (e.g., Scalasca), and binary analysis (e.g., HPCToolkit).

GTC has been the subject of previous performance studies (e.g. [21]) and the load balance issues discussed in this paper have also been identified in these earlier studies.

V. Conclusions

We have presented IPM, a modularized application performance and workload analysis tool for MPI, OpenMP and file I/O. Compared to the old implementation of IPM, which was focused only on MPI, we have extended the monitoring coverage into the areas of most importance for today's supercomputers (threading and file I/O). To the best of our knowledge IPM is the first research tool to utilize the compiler-inserted instrumentation for OpenMP monitoring. Previous solutions based on source-code instrumentation where not suitable for IPM's use case that mandates minimal user involvement.

IPM focuses on ease of use and scales today to several tens of thousands of processors. With our focus on lightweight, unintrusive measurement of performance we expect to be able to scale to even larger systems with minimal effort.

Our new modular design has allowed us to add measurement capabilities for OpenMP and I/O. It will also allow us to easily add measurement capabilities for new hardware features such as GPUs or other co-processors.

IPM's ease of use has enabled the NERSC center to collect over 310 000 profiles of jobs running 20 minutes or longer on three of its systems over the past six years to gain valuable insights into user and application behavior. Likewise, users benefit from the usage of IPM in their day to day code development and production runs by developing an understanding of resource requirements and tradeoffs. This is critical when dealing with varying "performance weather" on systems or when porting from one machine to another. Often an IPM profiling report can be the starting point for a more in depth evaluation with a tracing or statistical sampling tool.

In the transition to petascale computing, HPC will see shifts in the methods, libraries, and languages deployed by applications. IPM provides a modular way to adapt to the technology shifts in a way that reveals bottlenecks across all the layers involved. We hope that this vertical approach to performance analysis will provide an extensible and durable method to make HPC codes run faster. We plan to work on the addition of monitoring modules for a variety of data sources such as MPI-IO and network interface card counters. We are also currently working on a new parser and data presentation component targeted at supporting high-concurrency data.

Acknowledgements

This work was support by the NSF under award OCI-0721397. This research was supported by an allocation of advanced computing resources provided by the National Science Foundation. The computations were performed on Kraken (a Cray XT5) at the National Institute for Computational Sciences (http://www.nics.tennessee.edu/)

References

[1] "The Top 500 Supercomputer Sites, web page: http://www.top500.org."
[2] D. Skinner, "Integrated Performance Monitoring: A portable profiling infrastructure for parallel applications," in *Proc. ISC2005: International Supercomputing Conference*, Heidelberg, Germany, 2005.
[3] N. J. Wright, W. Pfeiffer, and A. Snavely, "Characterizing parallel scaling of scientific applications using IPM," in *The 10th LCI International Conference on High-Performance Clustered Computing*, March 10-12, 2009.
[4] G. Nakhimovsky, "Debugging and performance tuning with library interposers," Jul. 2001, http://developers.sun.com/solaris/articles/lib_interposers.html.
[5] B. Mohr, A. D. Malony, H.-C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger, and S. Shah, "A performance monitoring interface for OpenMP," in *Proceedings of the Fourth Workshop on OpenMP (EWOMP 2002)*, Rome, Italy, Sep. 2002.
[6] B. Mohr, A. D. Malony, S. S. Shende, and F. Wolf, "Towards a performance tool interface for OpenMP: An approach based on directive rewriting," in *Proceedings of the Third Workshop on OpenMP (EWOMP'01)*, Sep. 2001.
[7] 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 Conference on Supercomputing (SC 2003)*, Phoenix, Arizona, USA, Nov. 2003.
[8] S. Binet, F. Winklmeyer, W. Wiedenmann, P. Calafiura, and S. Snyder, "Harnessing multicores: Strategies and implementations in ATLAS," in *Proceedings of the 17th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2009)*, Prague, Czech Republic, 2009.
[9] "PAPI web page: http://icl.cs.utk.edu/papi/."
[10] Z. Lin, S. Ethier, T. Hahm, and W. Tang, "Size scaling of turbulent transport in magnetically confined plasmas," *Phys. Rev. Lett.*, vol. 88, 2002.
[11] A. D. Malony and S. S. Shende, "Performance technology for complex parallel and distributed systems," pp. 37–46, 2000.
[12] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications, ACTS Collection Special Issue*, 2005.
[13] N. R. Tallent, J. Mellor-Crummey, L. Adhianto, M. W. Fagan, and M. Krentel, "Diagnosing performance bottlenecks in emerging petascale applications," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–11.
[14] N. R. Tallent and J. M. Mellor-Crummey, "Effective performance measurement and analysis of multithreaded applications," *SIGPLAN Not.*, vol. 44, no. 4, pp. 229–240, 2009.
[15] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr, "Scalable parallel trace-based performance analysis," in *Proceedings of the 13th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI 2006)*, Bonn, Germany, 2006, pp. 303–312.
[16] Z. Szebenyi, B. J. N. Wylie, and F. Wolf, "Scalasca parallel performance analyses of PEPC," in *Proceedings of the Workshop on Productivity and Performance (PROPER 2008) at EuroPar 2008*, Las Palmas de Gran Canaria, Spain, 2008.
[17] "Using Cray performance analysis tools. http://docs.cray.com/books/S-2376-41/S-2376-41.pdf."
[18] "Intel Trace Analyzer http://www.intel.com/software/products/cluster/tanalyzer/."
[19] "Intel Thread Profiler http://www.intel.com/software/products/threading/tp/."
[20] K. Fürlinger and M. Gerndt, "ompP: A profiling tool for OpenMP," in *Proceedings of the First International Workshop on OpenMP (IWOMP 2005)*, Eugene, Oregon, USA, May 2005.
[21] G. Marin, G. Jin, and J. Mellor-Crummey, "Managing locality in grand challenge applications: a case study of the gyrokinetic toroidal code," *Journal of Physics: Conference Series*, vol. 125, no. 1, 2008.