Chapter 3
Performance Analysis and Workload Characterization with IPM

Karl Fürlinger, Nicholas J. Wright, and David Skinner

Abstract IPM is a profiling and workload characterization tool for MPI applications. IPM achieves its goal of minimizing the monitoring overhead by recording performance data in a fixed-size hashtable resident in memory and by carefully optimizing time-critical operations. At the same time, IPM offers very detailed and user-centric performance metrics. IPM's performance data is delivered as an XML file that can subsequently be used to generate a detailed profiling report in HTML format, avoiding the need for custom GUI applications. Pairwise communication volume and communication topology between processes, communication time breakdown across ranks, MPI operation timings, and MPI message sizes (buffer lengths) are some of IPM's most widely used metrics. IPM is free and distributed under the LGPL license.

3.1 Introduction

Performance analysis and workload characterization serve individual developers and computing centers to understand the performance characteristics of applications. Some important goals of this process are the identification and elimination of performance bottlenecks as well as the development of an understanding of how

Karl Fürlinger
Computer Science Division, EECS Department
University of California at Berkeley
Soda Hall 515
Berkeley, California 94720, USA
e-mail: fuerling@eecs.berkeley.edu

Nicholas J. Wright, David Skinner
NERSC Center
Lawrence Berkeley National Laboratory
Berkeley, California 94720, USA
e-mail: {deskinner, njwright}@lbl.gov

M.S. Müller et al. (eds.), *Tools for High Performance Computing 2009*,
DOI 10.1007/978-3-642-11261-4_3,

an application scales as the number of processing elements is increased. This paper describes the Integrated Performance Monitor (IPM), an MPI profiling and workload characterization tool that has very low overhead yet is able to deliver important user-centered metrics in detail.

The rest of this paper is organized as follows: In Sect. 3.2 we provide an overview of IPM's hashtable based monitoring approach. In Sect. 3.3 we describe how IPM is used to monitor an application and which kinds of performance data are delivered by IPM. In Sect. 3.4 we study the scalability of MILC (a quantum-chromodynamics code) with IPM. We discuss related work in Sect. 3.5 and conclude with an outlook on planned features and enhancements for IPM in Sect. 3.6.

3.2 Overview

We assume the following general model of an MPI application for the purpose of performance monitoring: The application is composed of *n processes*, each identified by an integer in $[0,\ldots,n-1]$, its *rank*. A set of events $E_i \subseteq E$ happen in each process i. We do not further formally specify what the events are, but we assume they occur at a certain time and have duration. Each event e has an associated *signature* $\sigma(e) \in S$ which captures the characteristics we are interested in. $\sigma : E \mapsto S$ is the signature function. Concretely we think of a signature $\sigma(e)$ as a k-tuple $\sigma(e) = (\sigma^1(e), \sigma^2(e), \ldots, \sigma^k(e))$, where each $\sigma^j()$ is a signature *component*. Useful components of signature functions are listed in Fig. 3.1.

Signature			Typical Size
Component	**Function**	**Data type**	**(#bits)**
Wallclock time	$time(e)$	floating point	32/64
Sequence number	$seq(e)$	integer	32
Type of MPI call	$call(e)$	integer	8
Message data size	$size(e)$	integer	32
Message data address	$address(e)$	integer	64
Message tag	$tag(e)$	integer	32
Own rank	$rank(e)$	integer	32
Partner rank	$partner(e)$	integer	32
Callsite ID	$csite(e)$	integer	16
Program region	$region(e)$	integer	8

Fig. 3.1: Components of an event signature function.

Our goal for performance observation is to get an event inventory of an application (i.e., understand the events that happened and their characteristics) by associating performance data (number of occurrences, statistics on the duration) with event signatures. If the signature includes $time()$, we essentially have a model for event tracing because the chronology of events can be reconstructed from the stored

signatures. If $time()$ and $seq()$ are not included in the signature, we have a model for profiling.

IPM is a profiling tool and for efficiency reasons we would like to keep the signature space much smaller than the event space ($|E| >> |S|$). In this case, the signature function will not be injective in general (many events can have the same signature) and performance data can be envisioned as a table indexed by the signature, with a number of columns for the statistics we are interested in. In IPM we implement this indexing using a hashtable resident in memory. The hash keys are 64 or 128 bits long and the timing statistics consume approximately 20 bytes per entry.

Upon program termination IPM writes a banner report and a log file in XML format. The banner contains the most important data about the program run in ASCII text and the log file provides the full information contained in the hashtable. In a post-processing step the XML is parsed and a profiling report is generated in HTML format. The contents of IPM's banner information and the full profiling report are discussed in the next section.

3.3 Performance Analysis with IPM

The most basic output IPM provides is a banner containing essential metrics which is written immediately after program termination. An example banner is shown in Fig. 3.2.

```
##IPM##################################################################
#
# command   : ./su3_rmd
# host      : nid03510                    mpi_tasks : 1024 on 86 nodes
# start     : Fri Nov 27 14:40:15 2009    wallclock : 20.57 sec
# stop      : Fri Nov 27 14:40:35 2009    %comm     : 55.69
# mem [GB]  : 108.98                      gflop/sec : 496.92
#
#           :      [total]         <avg>          min          max
# wallclock :     20961.67         20.47        20.42        20.57
# MPI calls :   1567890443       1531143      1531143      1531154
# MPI time  :     11673.29         11.40        11.26        11.94
# MPI [%]   :                      55.69        55.05        58.05
# mem [GB]  :       108.98          0.11         0.09         0.11
#
######################################################################
```

Fig. 3.2: The default banner provided by IPM upon program termination contains a number of important high level metrics.

The banner provides general information about the executed job, such as the start and stop date and time, the number of MPI processes used and the number of

different SMP nodes these processes where executing on. The next entries are the wallclock duration of the job in seconds and the percentage of overall time spent in MPI calls (`%comm`). If IPM has been installed on a system where these metrics can be acquired it also provides the total memory used by the application (`mem [GB]`) and the achieved floating point rate (`gflop/sec`).

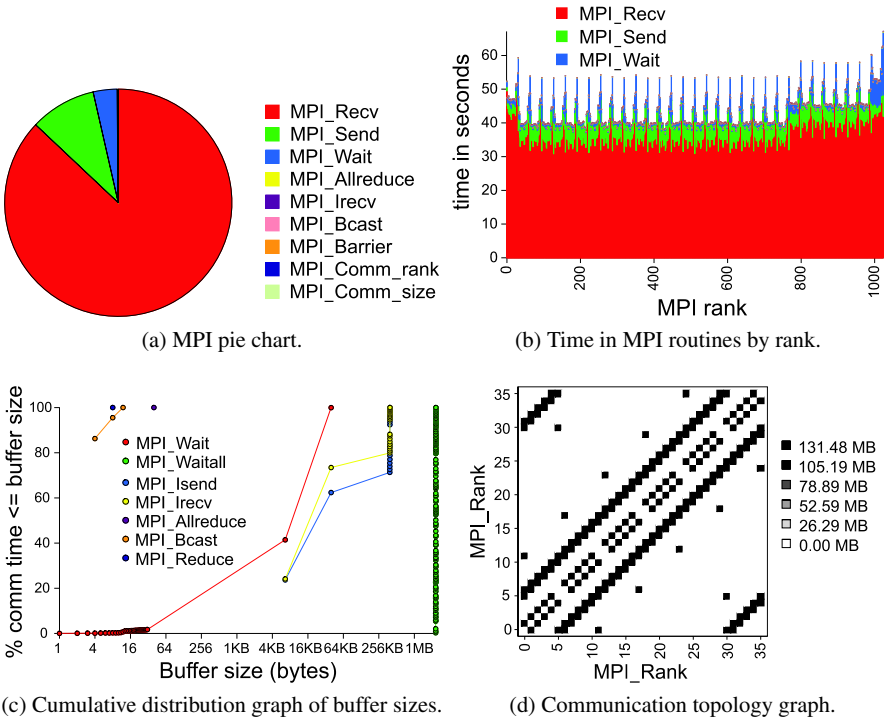
The lower part of the banner provides information about the distribution of key metrics across the MPI ranks. For each of wallclock execution time, the time in MPI calls, the number of MPI calls, the percentage of time in MPI and the DRAM memory used, this section provides the sum, average, minimum and maximum values.

A further section of the banner (not shown) is included if a full banner is requested by setting the `IPM_BANNER` environment variable to `full`. This section provides a listing of the individual most time-consuming events stored in the hashtable, and their distribution over the MPI ranks.

In addition to the text-based banner, IPM writes the full profiling data to an XML-based log file. Parallel MPI file I/O is used at high concurrencies to speed up the creation of this log file. A parser script is provided with IPM that reads the XML log file and produces an HTML page of the full profiling report which also includes charts and graphs to visualize the data.

Among others, the HTML profiling page contains these entries:

- The information contained in the text-based banner is reproduced in a table on top of the profiling report.
- A pie chart (Fig. 3.3a) 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 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 time) is shown in Fig. 3.3b. This type of display is especially useful to identify and locate load imbalance situations.
- Cumulative distribution graphs as the one shown in Fig. 3.3c 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.
- A communication topology graph as shown in Fig. 3.3d. 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.



(a) MPI pie chart.

(b) Time in MPI routines by rank.

(c) Cumulative distribution graph of buffer sizes.

(d) Communication topology graph.

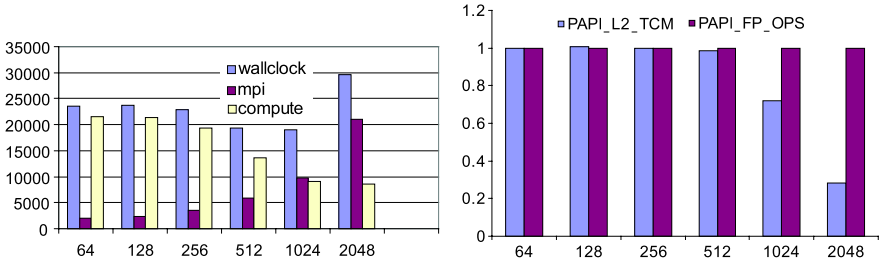
Fig. 3.3: Some of the performance data displays provided by IPM.

3.4 Example Scaling Study with IPM

This section describes an example scalability study using IPM. The application studied is MILC [6], a code to compute properties of elementary particles subject to the strong force as described by quantum chromodynamics (QCD). The computational domain is a 4D space-time lattice. Most of the computational work is spent doing a conjugate gradient inversion of the fermion Dirac operator. This entails repeated calls to a kernel that multiplies a large sparse matrix by a vector. Communication is mostly from point-to-point exchange of ghost-cell data on the 4D lattice. MILC was run with the medium problem size from the NERSC SSP (Sustained System Performance) benchmark set [8]. It performs 1375 conjugate gradient iterations on a 32^4 lattice.

The scaling study was performed on the Kraken Cray XT5 supercomputer at the National Institute for Computational Science (NICS) in Oakridge, Tennessee. At the time of writing this article Kraken was a dual-socket, 6-core AMD Opteron machine (12 cores total per node). Each processor is clocked at 2.6 GHz has 512

KB private second level cache, 6 MB shared L3 cache and each node has 12 GB of main memory.



(a) Scaling of MILC.

(b) Relative hardware counter scaling.

Fig. 3.4: MILC scaling study on Kraken.

We ran the program with 64, 128, 256, 512, 1024, and 2048 MPI processes and the graph in Fig. 3.4a shows the summed wallclock time (leftmost bar) for each run. Since this is a strong scaling study (the workload remains constant as the concurrency is increased), ideal scaling would be represented by constant summed wallclock times. For MILC, the summed wallclock execution time stays almost constant at 23,000 seconds for 64, 128, 256 processes, then drops to about 19,000 seconds for 512 and 1024, and then increases to about 30,000 seconds at 2048 processes.

Fig. 3.4a also shows the breakdown of the overall wallclock time into time spent in MPI operations and "compute" time (i.e., wallclock - MPI time). This breakdown sheds light on the nature of the wallclock scaling. Evidently the time in MPI (middle bar) increases dramatically as we increase the concurrency, from 2,016 seconds at 64 processes to 21,000 seconds at 2048 processes, but this increase is masked by a decrease in computation time (rightmost bar), most notably from 256 to 512 to 1024 MPI processes.

The most likely explanation for this super-linear speedup is the increased overall cache size at high concurrencies. At some point the data set will fit into the caches and cause less cache misses and traffic on the memory subsystem, leading to shorter execution times than could be expected from just the increase of the computing capacity alone. To test this hypothesis for MILC, we configured IPM to collect hardware counter events using PAPI [9], [1]. We measured the total number of floating point instructions (`PAPI_FP_OPS`) and the total number of level 2 cache misses (`PAPI_L2_TCM`). Figure 3.4b shows a chart of the relative scaling of the measured counter values as we increase the number of MPI processes. All values are normalized to the measured value at 64 processes. The number of floating point operations stays constant, since the same amount of work is performed in each run but the number of L2 cache misses decreases dramatically at 1024 and 2048 processes as the data set starts to fit the L2 cache.

By analyzing some of the basic metrics provided by IPM we were able to understand the scaling of MILC to a first order by separating out the contributions of messaging and computing. The next step would be to analyze the contribution of individual MPI calls (collective and point-to-point) and to consider how the message size distribution is changing and how the application might shift from being bandwidth bound to being latency bound. Both goals can be achieved readily using information provided in IPM's HTML profiling report.

3.5 Related Work

There are a number of other performance analysis tools, both employing tracing and profiling measurement techniques. Vampir [7], [2] is a trace collector and a trace visualizer for MPI and OpenMP applications. TAU [5], [11] is an extensive toolset for profiling and tracing of MPI/OpenMP applications and some other programming models. Automated performance analysis is the focus of other recent tools such as Periscope [3], [10] KOJAK [12], and Scalasca [4].

Compared to most of the aforementioned tools, IPM is focused more on giving the user a number of key metrics in a straightforward and easy way with low overheads and it places less emphasis on a detailed drill down into the application structure.

3.6 Conclusion and Outlook

We have introduce IPM, an Integrated Performance Monitoring framework with very low overhead. IPM can be used to derive essential key metrics for application characteristics such as percent of time spent in communication operations, imbalances in program regions or MPI calls, and the communication topology in a straightforward way.

For the path ahead, several improvements for IPM are under active investigation and development. We plan to add threading support to IPM for monitoring OpenMP and Pthreads applications. For file I/O a vertically integrated monitoring stack will allow us to monitor file operations at several layers (from the user's view to the underlying networking fabric) and will enable us gain novel insight into the interaction of storage, networking, and computation.

References

1. Shirley Browne, Jack Dongarra, N. Garner, G. Ho, and Philip J. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, 2000.

2. Holger Brunst and Bernd Mohr. Performance analysis of large-scale OpenMP and hybrid MPI/OpenMP applications with VampirNG. In *Proceedings of the First International Workshop on OpenMP (IWOMP 2005)*, Eugene, Oregon, USA, May 2005.
3. Karl Fürlinger and Michael Gerndt. Periscope: Performance analysis on large-scale systems. *InSiDE – Innovatives Supercomputing in Deutschland (Featured Article)*, 3(2, Autumn):26–29, 2005.
4. Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd 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)*, pages 303–312, Bonn, Germany, 2006.
5. Allen D. Malony and Sameer S. Shende. Performance technology for complex parallel and distributed systems. pages 37–46, 2000.
6. MILC website, `http://physics.indiana.edu/~sg/milc.html`.
7. Wolfgang E. Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–90, 1996.
8. Sustained system performance benchmarks at NERSC, `http://www.nersc.gov/projects/ssp.php`.
9. PAPI web page: `http://icl.cs.utk.edu/papi/`.
10. Periscope project homepage `http://wwwbode.cs.tum.edu/~gerndt/home/Research/PERISCOPE/Periscope.htm`.
11. Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications, ACTS Collection Special Issue*, 2005.
12. Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. In *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003)*, pages 13–22. IEEE Computer Society, February 2003.