REMORA: a Resource Monitoring Tool for Everyone

Carlos Rosales
Texas Advanced Computing
Center
University of Texas at Austin
Austin, TX
carlos@tacc.utexas.edu

Antonio Gómez-Iglesias
Texas Advanced Computing
Center
University of Texas at Austin
Austin, TX
agomez@tacc.utexas.edu

Andrew Predoehl
Department of Computer
Science
University of Arizona
Tucson, AZ
predoehl@email.arizona.edu

ABSTRACT

Knowing about the requirements of HPC applications is a common question that users of high performance systems ask often. However, answering this question requires the collaboration of administrators and sometimes the answer does not contain the amount of detail that users demand. This work introduces a new user space resource monitoring tool, REMORA. REMORA stands for REsource MONitoring for Remote Applications, and provides a simple interface to gather important system utilization data while running on HPC systems. REMORA is designed to provide a brief text report and post-processing tools to analyze the very detailed records taken during an application run. Users can configure the tool to achieve the amount of detail that they want and perform the analysis of the results at any point in time. REMORA helps users achieving a better understanding of their applications by providing a high level profile of their executions and users can take advantage of that information to improve their codes.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—*diagnostics, distributed debugging*; D.2.8 [Software Engineering]: Metrics—*complexity measures, performance measures*

1. INTRODUCTION

Most users of HPC systems are scientists and domain specialists who might only have a superficial knowledge of the hardware and system configuration used in their daily work. Many system configuration details can affect the ability of users to execute their workloads successfully, but it is often difficult for users to determine the detailed interactions of their programs with the system.

Issues regarding memory footprint, I/O (Input/Output) load, and network mapping represent the bulk of the problems that are addressed regularly by support staff at centers like the Texas Advanced Computing Center (TACC). For

example, in the period between July 1st 2014 and July 1st 2015 approximately 18% of over 5000 support questions at TACC involved memory or file system utilization issues.

A typical user may want to know how much additional I/O load his code will generate when certain options are chosen in the configuration file for a large run, but without the tools to determine the same data for smaller cases and alternative configurations the prediction is impossible. Similarly, a user that experiences execution crashes when the operating system kills his job due to excessive memory usage may want to investigate a time series of detailed memory utilization, but without a tool that provides this information trial and error is the only option.

The situation gets compounded in heterogeneous systems, where information associated with execution on an accelerator or coprocessor may be difficult to obtain but critical for the optimal setup of the user workflow.

While there are other tools that already collect many different statistics about the jobs that run in HPC systems [6], in most cases users want very simple data with some high degree of detail and immediate availability. In those cases, system wide automated collection tools typically fail to capture most of the details since they try to have a very low overhead as they run for every single job that is executed in the system. The results these tools produce are comprehensive in nature and often difficult to understand, and their output is often collected in a centralized database with access limited to staff, which prevents direct access by the user and delays analysis.

Also, there are other tools [4] that help users to collect information about how much memory their jobs require. However, tools like the one here cited do not provide enough information that users need when profiling their applications. Based on the needs of HPC users, we want a tool that can provide information in real-time, so that users can monitor how the application is using the computational resources while the application is running.

This calls for a tool that can provide to any user a significant amount of information regarding runtime resource utilization with minimal execution overhead. The tool should also provide some level of analysis and facilitate an easy visualization of the statistics collected while the application was running. This information should also be useful for system administrators as well as other people involved in the day-to-day operations of clusters since it might provide additional insights on why an application is not performing as expected or why a given job created issues in the machines. All these elements have been combined into a new

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HUST2015 November 15-20, 2015, Austin, TX, USA

© 2015 ACM. ISBN 978-1-4503-4000-7/15/11...\$15.00

DOI: <http://dx.doi.org/10.1145/2834996.2834999>

tool (REMORA), available to all the users of Stampede as a module.

Although REMORA is not designed to run with every job, it has been designed to perform the collection of the data in a way that minimizes the impact in the overall execution. There are many performance and resource monitoring tools, but we must stress that the role of REMORA is not to substitute those but rather complement them by providing an easily understandable summary of the resource utilization. REMORA has been designed with the aim of being easy to use by any HPC user. The benefits of the tool are varied, from detecting bugs in the code to giving additional insights on the performance of the code. Many users can take advantage of this tool to improve their usage of HPC resources.

The rest of this paper is organized as follows: Section 2 details the implementation of the REMORA tool. In Section 3 we show different examples of different applications and how REMORA helps to detect issues on them. Finally, in Section 4 we summarize the main contributions of the paper and propose future extensions for this tool.

2. DESIGN AND IMPLEMENTATION

A critical point in the design of the tool is the ease of use. The goal is to create a usage model that users can take advantage of with minimal effort. While there are models where the tool could run transparently, just by loading a module that would change the environment to collect all the information at runtime, we decided to opt for a model where the user loads the module, and then changes the submission script to invoke our tool by prepending its name to the actual command that has to be analyzed. This has the advantage of preventing unnecessary overhead when the module is loaded and runs that do not require instrumentation are submitted. It also simplifies the data collection for serial jobs and scripts, which do not use an MPI launcher that can be easily hijacked or modified. For example, if the original command is `mpirun ./myparallelapp`, the new command will be `remora mpirun ./myparallelapp`. A more complicated scenario is when the user wants to run a set of different scripts in the same job. In that case, the user will have to put all the commands regarding the execution in a shell script (i.e. a shell script called `myjob`). Then, all the user has to do is to call the script as follows `remora ./myjob`. The tool can be used in a batch script or interactively in the command prompt.

In its current implementation the tool generates a flat ssh tree with a single connection from the master node in the execution to all other nodes. This connection initiates a background process that collects the statistics for each node with a frequency specified by the user. The remote background tasks execute a loop over all processes owned by the user, and aggregate the data before committing it to file.

As previously stated, REMORA allows users to monitor the statistics as they are being generated. All the collected data is written to files accessible why the users in real time.

2.1 Profiling Overhead

Two types of overhead must be considered for any tool that provides real time resource monitoring. The first one is a setup overhead, independent of the frequency used to poll system resource utilization, and corresponds to the sum of

checks performed only before and after the code runs and the time that it takes to establish remote connections and launch the data gathering tasks. The second type of overhead corresponds to the delay in executing the code when a background task is sharing hardware resources. This second type of overhead depends on the frequency of system resource polling, and we will refer to it as a runtime overhead. Overhead studies were made with a Lattice Boltzmann Method code [9, 2], but results should be similar for other applications. It should be noted that in the rare case that a compute-bound application uses high frequency resource polling the overhead may be higher than those presented here.

As shown in Figure 1 the setup overhead for REMORA increases strongly with the number of nodes used in the run. This is to be expected because the current implementation uses a flat ssh tree to launch remote processes. A solution to this particular issue, which the authors plan to implement in the near future, is to spawn the remote tasks using a multi-level tree startup procedure. Other data capturing issues, such as the network topology information, are also relatively expensive to obtain. To keep things in perspective, however, it is important to remember that this tool is designed to help debug runtime issues, and in that context a few seconds of setup overhead are typically worth the data provided by the tool.

Figure 1: Setup overhead as a function of job scale for CPU only (CPU) and symmetric execution (SYM).

Runtime overheads are different for jobs that run only on CPUs and jobs that run on both CPUs and Intel Xeon Phi co-processors. In the worst-case scenario, where all cores are used in a node, REMORA’s runtime overhead remains well below 10% of the original application runtime for pure CPU execution. In the case of symmetric execution (1 MPI task per host node, 2 MPI tasks per Phi co-processor) the runtime overhead also remains below 10% of the original runtime. The smoother behavior in symmetric mode is, in part, due to the REMORA collection tasks being bound to core 61 on the Xeon Phi co-processor, which is not used by the application. As shown in figures 2 and 3 the runtime overhead depends very weakly on the scale of the job.

2.2 Execution Customization

REMORA is configurable in terms of the amount and type of data collected, but sensible defaults are provided to sim-

Figure 2: Runtime overhead for CPU only jobs as a function of the job scale for different polling periods.

Figure 3: Runtime overhead for symmetric jobs as a function of the job scale for different polling periods.

ply its use. By default the statistics are collected every ten seconds.

REMORA provides two different running modes and it also allows the users to specify how frequently the data is collected. The behavior of the application is controlled via two environment variables:

- **REMORA_PERIOD:** Time in seconds between consecutive data records. This is the time from the end of a collection event until the start of the next collection event. Depending on the platform where the tool is running, the overhead introduced by the application can make the duration of the collection event to vary, in which case there will be less data points in the collected results than expected. However, in the systems that we have tested the overhead of the application is small enough that the total number of collection points (CP) is almost equal to $CP = ET/RP$ where ET is the execution time (in seconds) and RP is the period (in seconds).
- **REMORA_MODE:** this variable accepts two possible values (FULL and BASIC). The FULL mode runs all the tests that the tool allows, as described in 2.3. The

BASIC mode only reports memory and CPU usage. This is the recommended mode when the users now that the application of interest does not create problems in the distributed file system. The tool that checks for the file system load [5] is the main contributor to the small overhead introduced by the tool. When this report is not required, the overall overhead of the tool is very small, as shown in Section 2.1.

2.3 Statistics Collected

REMORA collects a set of statistics that are useful in many different scenarios when profiling an application. The data collected by REMORA consists of:

- Detailed timing of the application.
- CPU utilization.
- Memory utilization.
- NUMA information.
- I/O information (file system load and Lustre traffic).
- Network information (topology and InfiniBand traffic).

Dynamic information is collected every REMORA_PERIOD seconds. The following describes the data collected in more detail.

CPU.

The application reports the average CPU usage of the last second (independently of the value specified for REMORA_PERIOD). This information is very important for applications that use OpenMP, where it is possible to easily analyze how the cores are being used. It also allows to check for a correct pinning of threads to the cores: not pinning processes could lead to threads floating between cores, which will be show up in this report. MPI applications can also benefit from this information.

Memory.

One of the most recurring questions for HPC users is “how can I know how much memory my job is using?”. Trying to answer that question, REMORA collects the most relevant statistics regarding memory usage every REMORA_PERIOD seconds:

- **Virtual Memory (and Max Virtual Memory):** this is a very important value as the OOM (out of memory) killer will use it to kill the application if needed.
- **Resident Memory (and Max Resident Memory):** physical memory used by the application.
- **Shared memory:** applications have access to shared memory by means of /dev/shm. Any file that is put there counts towards the memory used by the application, so the application reports this usage.
- **Total free memory:** this will take into account the memory not being used by the application, the libraries needed by the application, and the OS.

Data is collected from `/proc/<pid>/status` for all of these except shared memory, which can be obtained from `/dev/shm`. Memory usage for all user processes is aggregated and written to a single file per node involved in the execution. At the end of the run the maximum values for memory utilization (and minimum value of total free memory) are aggregated into a single file.

When Xeon Phi co-processors are used as part of a symmetric execution model, each Xeon Phi is treated as a separate node and the same memory information is collected from Phi and from host CPU. Individual files are maintained for each node and Phi and the per node aggregated summary is provided individually for nodes and Xeon Phi devices, since their available memory tends to be different.

NUMA.

As it is well known, NUMA (Non-Uniform Memory Access) can have a large impact on the overall performance of an application. Sometimes small changes in the code can lead to large improvements once it has been discovered that NUMA was imposing a penalty over the application. Our tool reports how memory is being used in each socket and it also collects the number of NUMA hits and misses. The information is extracted from the `numastat` tool: `numastat` is called only once on each collecting period; the output of `numastat` is then analyzed and several different fields are used for the statistics:

- Number of hits: total number of memory access hits.
- Number of misses: total number of memory access misses.
- Number of hits in the current node: if the data that the application was looking for is in the same of node where the core requesting that data is located, it produces a hit in the current NUMA node.
- Number of hits in the other node: when the data required is in cache, but in the other NUMA node.
- Total memory free/used on each node.

xltop.

`xltop` [5] is a Lustre monitor developed at TACC that can be freely downloaded and installed. It provides statistics about file system utilization, including I/O requests per second and read and write volumes in MB/s. Originally, `xltop` was a continuous monitoring client that connects to other tools (`xltop-master`, `xltop-servd`, and `xltop-clusd`) to retrieve the information statistics from every MDS (meta-data server) and OSS (Object Storage Server). This is a tool used by administrators to specifically look for very heavy I/O applications that can damage the stability of the file system. We have modified the original `xltop` code so that it can be invoked by any user to only check for the Lustre I/O statistics of the job being analyzed. Also, while `xltop` originally used `ncurses` to show the results, we have now implemented a version that collects the statistics in a file in order to simplify the analysis of the results.

Lustre Stats.

Our tool collects information regarding the data transferred by Lustre on each node used by the job while running.

Normally, these statistics do not provide much information to the users. However, they are very useful if there was a problem in the file system while the job was being analyzed, as the number of messages dropped will significantly increase. The following Lustre information is collected:

- Number of currently active Lustre messages. It also includes a highwater mark of this value.
- Messages sent/received: total number of Lustre messages sent/received by the current node.
- Messages dropped: number of Lustre messages that failed to be delivered to the destination.
- Bytes sent/received: total number of bytes sent/received on Lustre messages.

InfiniBand Packets.

Number of packets transmitted using InfiniBand. This data can be used to get extra information regarding how the communication in parallel applications takes place. In particular, the time series can be used to correlate high network activity levels with sections of the code, and those sections can be revised for possible optimization.

2.4 Post-Processing

All the data is collected in a set of files with the statistics organized in columns. Users can take those files and run any postprocessing tool that they develop. However, for simplicity, REMORA already provides a plotting script that takes all the statistics generated during collection and generates a number of plots. These plots show the most relevant information previously collected and represent a visual alternative while analyzing the results. This is a Python script that requires NumPy, matplotlib and blockdiag. The script can be called from the batch script or from the login nodes after the job has finished. In this second scenario, the script requires the id of the job to be analyzed as argument to distinguish among different results that might have been previously collected.

2.5 Stampede Installation

We have created a module called *remora* and installed it on Stampede. Stampede uses Lmod [1] to help users to control their environment. This module is a core module in Lmod, independent of compilers, so that all users can load it independently of their environment.

3. CASE STUDIES

In this section we present different scenarios where REMORA has proven to be a useful tool in analyzing the behavior of existing applications and common use cases.

3.1 Example of Memory Usage

OpenMX is a software package for ab-initio materials simulation based on the Density functional Theory (DFT) [7, 8] with an advanced MPI+OpenMP hybrid implementation [3]. One of the researchers running OpenMX at TACC was making use of the large memory nodes to run a relatively small memory footprint job, which used only about 60GB out of the available 1TB of memory per node.

These large memory nodes are in high demand, and the user was contacted and asked if he could run on a larger subset of regular nodes (limited to 32 GB/node). The user attempted running on the regular nodes but his jobs crashed, and the automated data monitoring system was only able to report a 4GB memory usage due to its coarse polling time (10 minutes, while the application crashed in 7). Using REMORA to analyze the memory usage in one second intervals it became clear that the virtual memory footprint of the code was too large to be run on the number of nodes it had been assigned, since the Out Of Memory killer was invoked when memory usage per node went over 30.5 GB/node. The corresponding memory profile obtained is shown in Figure 4.

A few short additional test runs determined that the user could run in the regular nodes provided not more than 4 MPI tasks were spawned on each node, freeing a significant portion of the large memory nodes for other users. Thanks to the quantitative data provided by REMORA this analysis was completed in a few hours, including the execution of the jobs.

Figure 4: Memory usage on master node for a failed OpenMX execution.

3.2 Symmetric Execution

Stampede was the largest Intel Xeon Phi deployment when it came online in 2013, and the co-processor component constitutes nearly three quarters of its approximately 10 PFlop capacity. It is important to provide users with as much information as possible regarding execution in the co-processors, since their memory footprint and compute capabilities are very different from those of the host CPUs. REMORA provides a simple means of tracing memory utilization in both host and co-processor, in a manner transparent to the user.

As an example, a Lattice Boltzmann Method (LBM) code [2] was run in symmetric mode, using one MPI task on the host CPU and two MPI tasks on the Phi co-processor. A total of 16 OMP threads per task on the host and 120 OMP threads per task on the Phi were used. Figure 5 shows the memory used in both host and co-processor for the duration of the run. The information provided by this figure is valu-

able because it gives a direct indication of how much larger a case could be run on the Phi before exhausting available memory.

Figure 5: Memory usage in a symmetric execution run.

3.3 MPI Applications

REMORA will collect the statistics previously described for all the nodes used in a parallel job. This information allows to discover load imbalances in the computation, as different nodes might have very different CPU or memory loads. For the case of well-balanced applications the information collected should not differ significantly, as shown in Fig. 6 and 7. It will also collect information about the network and how the nodes involved in the computation are connected, as shown in Fig. 8 (this information is useful for systems administrators, who can use it to find problematic links in the network).

Figure 6: Memory usage in the master node on an MPI application.

3.4 OpenMP Applications

3.4.1 Parallelization

Figure 7: Memory usage for other nodes of the MPI application.

Figure 8: Network diagram of an MPI job. Core and leaf switches are shown.

A common question from users using OpenMP is that their code is not performing as expected, or it is not using all the cores available in each node even after setting the correct value for `OMP_NUM_THREADS`. Typically, in Stampede, it means that the user has not specified any affinity when the code is compiled with Intel compilers (`KMP_AFFINITY`). When affinity is not set, the CPU utilization is highly variable as threads jump from one core to another during the execution, as shown in Fig. 9.

However, after setting the affinity (to compact in this case), the CPU utilization is homogeneous and more efficient as shown in Fig. 10. It can be seen how this is a small and easy change for the users. It also ensures that the environment used by the users is controlled by them, so that the reproducibility of their results is better.

3.4.2 Data Initialization

In machines like Stampede, NUMA represents a key element to consider when trying to achieve the best performance possible. There are specific cases where NUMA statistics play a critical role when giving insights about issues that might be affecting the performance of an application.

Figure 9: CPU utilization with affinity not set. Not all the cores are fully used.

Figure 10: CPU utilization with affinity set.

These statistics are useful for example to identify problems regarding the first touch placement policy, which is critical in non-optimized OpenMP applications. Many users and developers do not think of this policy when initializing shared data. In this case, statistics like memory usage are not relevant. Fig. 11 shows the statistics regarding NUMA for a non-optimized OpenMP application. In this application the data is initialized outside of a parallel region, so that the master thread first touches all the shared data that will be later required by the other threads. It can be seen how only the memory in Node0, the socket where the master thread resides, is used. Threads that run on the other socket will have to use data allocated in the remote memory. In Fig. 12 the data initialization takes place in a parallel region. In this case, the memory in both sockets is used, and each thread will have its data in the closest memory. It is important to consider that the memory usage statistic will not be of much use in this case (Fig. 13) since it will show the same data for both implementations.

3.5 Applications Workflow

Figure 11: NUMA. Serial initialization.

Figure 12: NUMA. Parallel initialization.

Figure 13: NUMA. Total Memory Usage.

A common scenario in many scientific fields consists of running many different applications sequentially where the output of one application is used by the next one. We previously described how to use REMORA in this scenario. Many times, the problem in this workflow resides in the fact that only one of the components of the workflow has high requirements in for example, memory usage. This is the case of the example shown in Figs. 14 and 15. This is a bioinformatic workflow where most of the components of the workflow are sequential (or the amount of parallelization provided is very small). It can be seen how for the first half of the execution only one core is used for most of that time. This correspond to an application that only has a for loop parallelized (that explains the small peaks in the CPU usage). It can also be seen how the different applications being used in the workflow are assigned to different CPUs, as the user does not specify any pinning. It can also be seen how most of the memory is used by the main application of the workflow, but how the memory usage significantly varies during the entire execution of the workflow as different applications are running at each step.

Figure 14: CPU usage for a workflow. CPU utilization fluctuates as different applications are executed.

3.6 High I/O

Many applications implement I/O without considering that the approach used for those operations might not be the most efficient. In many cases, applications are converted from an original serial implementation to a parallel version, without redesigning the I/O appropriately. While a serial implementation running on the local disk of a workstation might not run into problems, the equivalent parallel implementation can lead to overloads of the metadata servers in a distributed file system. Typical examples are implementations where a file is opened and closed within a for loop just to read one single element of the file.

While developing REMORA a user was contacted because system logs indicated that his runs were producing excessive I/O traffic to the metadata servers. To confirm the origin of the issue, the execution of a single job was wrapped in REMORA and the I/O data recorded in one second intervals. Figure 16 shows that the execution had bursty I/O reaching 9500 requests per second. While a single run would

Figure 15: Memory usage for a workflow. This scenario usually follows an irregular pattern.

not affect the file system, multiple jobs happened to be running simultaneously and impacted the system. With the data in hand the developer was able of analyzing the code and determine the source of the I/O load. After modifying the code a second run was tested and shown a 15X reduction in I/O load. This not only allows running many instances simultaneously without generating load issues, but also reduces execution time by 22%.

Figure 16: I/O load. High load for the original code and reduced load after changes in the code were made.

4. CONCLUSIONS

In this paper we have presented a tool that will help HPC users to get better insights into the resource requirements of their applications. This tool allows to easily extract memory and CPU utilization, as well as more advanced statistics like I/O requests. This information is a typical request from users of these infrastructures.

The different statistics provided by REMORA allow all

type of applications to benefit from this tool. The data extracted can be used to detect problems in the codes and also to improve the performance of jobs that are already running.

We have presented different scenarios where the data collected by REMORA provides relevant insights of the main characteristics of the applications involved in those scenarios. Many more scenarios are possible, and we expect the tool to be able to handle them and offer valuable data to the users.

While several of these statistics are easily accessible for experienced users, some of the data provided - such as I/O details - was not previously available for regular users. REMORA now allows anyone to have access to this information. Applications can highly benefit from this particular data since users are usually unaware of high I/O loads in their codes.

Further work includes extension to monitoring GPU accelerators, as well as setup overhead reduction. We also plan on designing a plugin interface that will allow to easily extend the application to collect other statistics.

5. ACKNOWLEDGMENTS

The authors would like to thank UC Riverside user Kuan Zhou for his willingness to test our tool while still in development. Also, we would like to mention our colleagues Doug James, Cyrus Proctor, Antia Lamas and Si Liu for their valuable comments and suggestions. The authors also gratefully acknowledge National Science Foundation support under award numbers ACI-1134872 (Stampede), and ACI 10-53575 (XSEDE).

6. REFERENCES

- [1] Lmod. <http://sourceforge.net/projects/lmod/>. Accessed: 2015-08-11.
- [2] mplabs. <https://github.com/carlosrosales/mplabs/>. Accessed: 2015-08-11.
- [3] OpenMX. <http://www.openmx-square.org/>. Accessed: 2015-08-11.
- [4] peak_memusage. https://github.com/davidedelvento/peak_memusage. Accessed: 2015-08-11.
- [5] xltop. <https://github.com/jhammond/xltop>. Accessed: 2015-08-11.
- [6] T. Evans, W. L. Barth, J. C. Browne, R. L. DeLeon, T. R. Furlani, S. M. Gallo, M. D. Jones, and A. K. Patra. Comprehensive resource use monitoring for hpc systems with tacc stats. In *Proceedings of the First International Workshop on HPC User Support Tools*, HUST '14, pages 13–21, Piscataway, NJ, USA, 2014. IEEE Press.
- [7] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Physical review*, 136(3B):B864, 1964.
- [8] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Physical Review*, 140(4A):A1133, 1965.
- [9] S. Succi. *The lattice Boltzmann equation: for fluid dynamics and beyond*. Oxford university press, 2001.