Profiling MPI Applications with Mixed Instrumentation

Eduardo Grosclaude, Claudio Zanellato, Javier Balladini,
Rodolfo del Castillo, Silvia Castro

Universidad Nacional del Comahue; Dpto. de Cs. e Ing. de la Computación, Universidad Nacional del Sur
{oso,czanella,jballadi,rolo}@uncoma.edu.ar, smc@cs.uns.edu.ar

Abstract. Our research project intends to build knowledge about HPC problems to be able to help local researchers. In order to advise users in choosing parallel machines to run their applications, we want to establish a general methodology, requiring as shallow information as possible, to characterize parallel applications. To draw a profile of a closed, message-passing application, we look for convenient tools for inspection on the distribution of communication primitives. We show a feasible way to do black-box instrumentation of closed MPI applications.

Introduction

Our recently started research group at Universidad Nacional del Comahue aims at building knowledge about high performance computing (HPC) hardware and software resources, and help local researchers from other scientific and engineering fields to better understand and adopt them [2]. Our research project also intends to help some of its members pursue their postgraduate studies. The present work is a preliminary step towards the formulation of one Magister thesis in HPC, oriented to application performance prediction in hybrid environments.

As a first stage in our research, we want to build a general, shallow-knowledge, methodology to characterize parallel applications. We are studying performance analysis methods and techniques, and looking for ways to perform black-box profiling of parallel binaries. In the present work, we gather some results of our first forays into performance analysis and program optimization as they are actually done today, and show how we can use some techniques to profile applications without access to source code.

The present work begins with an overview of performance analysis concepts and techniques. Then, we analyze different ways to do application instrumentation for profiling and tracing, and we review some performance analysis tools currently in use. Later, we specifically consider instrumentation of MPI applications. We explain our mixed instrumentation technique, which enables us to do black-box profiling of parallel programs, and its applicability conditions. We present an overhead benchmark conducted upon this technique. Finally, we offer our conclusions and describe some future work we plan to take up.

Performance analysis

Writing parallel programs is usually a hard task, and results are not always acceptable at the first attempt. To help themselves optimize their parallel applications, programmers often iterate around a common, well-known cycle of instrumenting applications; running the parallel application to take measurements; introduce the information collected into some analysis and presentation tool; and then perform any optimizations suggested by the former stage. Instrumentation is the stage we are most concerned with, at this moment. Program instrumentation is the insertion of code into the program, to record points in time or in the program's text where interesting events happen. Instrumentation can be carried on by the programmer or by a tool, and can be done at several levels:

Source-level instrumentation. At the source level, programmers can modify source code to include event-recording sentences. This method is the most flexible of all but is also labor-demanding, requires recompiling, may clash with compiler optimizations and obviously needs access to source code.

Object-level. By using wrapper libraries, programs can reveal information about their use of function calls. Calls are intercepted by performance analysis tools which reimplement library functions, so they are able to do accounting on these call events. However, while control is inside such reimplemented functions, it may not be possible to trace back calls to the original source line in the program, thus making tuning difficult. Wrapper instrumentation usually works by relinking applications' object code to new libraries.

Binary-level. Binary level instrumentation is the insertion of event-recording code directly into the executable file or program image in memory. Both source- and object-level instrumentation techniques require some access to separate components of the application under study. Binary level instrumentation, on the other hand, can offer a pure black-box approach to performance analysis. However, at the binary level, events are not easily mapped back to source instructions. While convenient for a user, this kind of instrumentation is technically complex. It calls for a deeper knowledge of the underlying architecture and is not easily portable. Binary instrumentation can also be a powerful means for dynamic tuning [19,21].

Typical byproducts of instrumentation are profiles and traces. A program trace is a stream of program event records, usually with their timestamps and possibly some associated metrics. A program profile is the summarized information about a set of events verified during execution. While tracing is oriented to describing the whole lifetime of a program, profiling usually considers time-oriented metrics or number of events accomplished.

Events of interest for tracing or profiling in parallel programs may be related to machine-level activity, function calls inside the application, or calls to other libraries.

Hardware events Modern machines provide hardware counters. Selected machine events (such as cache hits or misses, TLB misses, or floating point operations) are counted by hardware registers. When these hardware counters overflow, or some threshold is met, they generate interrupts. Users can catch these interrupts to do any related work, such as tracing or accounting for profiling, with minimal intrusion on running applications.

Function calls An application can profile the usage of its own functions to reveal where it spends most of its execution time. Thus, a programmer who desires to optimize a program can get the most from her efforts by first dealing with the most time-consuming regions in the program.

Calls to other libraries Often, parallel programmers are interested in other calls, such as those related to calculations or communications. How many communication functions are called, and how long they take, can tell about scalability and bottlenecks in the application. MPI functions are natural candidates for tracing and profiling.

While tracing provides very detailed and deep information, traces can be very large and hard to work with, requiring special data structures and processing of their own [16]. Best performance analysis tools for dealing with traces are those which allow the user to visually and interactively place her queries. On the other hand, profiles are dense pieces of information which can quickly pinpoint bottlenecks or program zones where optimization can cause greater effects, but lack detail and do not provide a timeline view of the program. Profiles lend themselves better to statistical comparison among groups of runs or experiments. Some hybrid approaches intend to preserve the time-related information while keeping data at manageable sizes, such as incremental profiling, where the behaviour of an application is described by a sequence of snapshots.

Some example tools

Many performance analysis tools are currently used and supported. Surveys or rankings according to several criteria have been published [22,10,21,17]. Although portability is a frequent criterion, here we refer to some tools in the domain of Linux applications.

Machine-oriented tools Several tools exploit hardware event monitoring capabilities. Tools like oprofile [1] are standalone, while PAPI [18], Perfsuite [4], TAU [12], depend on lower layers such as implemented by perfctr [11]. While low-level instrumentation libraries exist since some time already, the in-kernel ability to use hardware counters appears only in recent Linux kernel versions [9]. Perfcounters is a substitute project to the long-time known oprofile as well. Perf, the tool designed for the perfcounters project, provides per task, per CPU and per-workload counters, counter groups, and sampling. Operating system events such as minor/major page faults, task migrations, or context-switches are also registered.

STRACE and GPROF. A common tracing program known to users of modern UNIX-like systems is STRACE. During a run under control of the strace program, the system calls issued by an application are listed, along with their timestamps, arguments and results. Several strace options allow the user to better define her queries, say, narrowing down the interesting calls to a family, or computing the time spent into every call. A well-known profiling tool is GNU GPROF [3]. After execution (*post-mortem*) of an application compiled with gprof support, the gprof tool outputs a call tree or ranks the program's functions by number of calls and time spent into each of them. Incidentally, gprof can be made to work with MPI applications. When the enviroment variable GMON_OUT_PREFIX is defined, each process is caused to write a profile describing its own execution under a unique name. The marginal profiles can be later merged or summarized by gprof to present a unique overall view.

VAMPIRTRACE and VAMPIR. VampirTrace is an MPI tracing facility bundled with the Open MPI library, implemented as a wrapper to the MPICC compiler plus an instrumentation library (LIBOTF). During execution, an application compiled with VampirTrace support generates a set of files including one trace file per process. The format for this set of files is known as the Open Trace Format (OTF) [13]. The framework comes with some simple utilities to exploit OTF traces. However, the only known graphical, interactive program able to manipulate OTF traces we could identify is Vampir, a commercial product [14]. Also, there is a notorious lack in conversion tools to or from other trace formats [20]. These are surprising facts, as the trace format specification and library are published as open source software under a non-restrictive license. Unlike MPE/Jumpshot, which are confined to MPI concepts, VampirTrace is more generally powerful, as it provides a mechanism for recording and analyzing not only MPI events but also those in other domains as machine events or OpenMP threads.

MPE, Jumpshot. MPE (Multi Processing Environment) comes bundled with the MPICH implementation of MPI [5], but can be used with other major MPI implementations. MPE is probably the most popular performance analysis framework for MPI because of the extended MPICH user base and the freely available JUMPSHOT visual tool. MPE is a link-time library but also provides a set of functions the programmer can use to develop her own tracing strategies. Jumpshot is a Java application which can effectively work with large traces. The trace format in MPE has undergone significant evolution (ALOG, BLOG, CLOG, SLOG, SLOG2...) to face scalability challenges.

mpiP. Proposed as a “lightweight, scalable MPI profiling” library [7,24], MPIP links with an object file, intercepting calls to MPI on a process basis. The results are merged at the end of the run, so no extra inter-process communication is required. The human-readable output identifies requests to MPI in the program (“callsites”), and summarizes the number of calls and traffic information for each callsite in several ways. When source files are compiled with debugging support, mpiP can correlate callsite statistics to source lines. The library source code is available.

MPI Instrumentation

We now consider the problem of MPI profiling, how it is actually done by current profiling libraries, and how to adapt them to our needs.

MPI Profiling Interface

The MPI definition [6] includes a mechanism to enable users to profile MPI applications. This mechanism is called MPI’s Profiling Interface Layer. The purpose of this interface is to enable easy development of profiling tools in a portable way. The interface does not state in any sense how this profiling should be done, nor is its use limited to profiling purposes.

To meet the MPI profiling interface, any implementation of the MPI library must contain a secondary entry point to the library’s routines. While, in the application programmer’s view, all MPI function names are prefixed by “MPI_”, secondary names for the same functions are prefixed by "PMPI_". When linking an application against a profiling library, any application's MPI-prefixed function calls can be transparently intercepted, and then PMPI-prefixed versions can be called (Figure 1 (a) and (b)). The intercepting function can do any other desired work such as event logging, event accounting, or notifying other applications. PMPI-prefixed function calls are exactly equivalent to the usual MPI-prefixed ones, so the only difference in the application behaviour should be some, hopefully small, overhead.

As the design of the MPI Profiling Interface is intended to be portable across implementations and platforms, the profiling interface does not require users to have access to the MPI code or implementation details. However, the user is normally required to, at least, relink her application against the profiling library. When building the instrumented application, the linker must find the object implementation of MPI-prefixed functions, provided by the profiling library, earlier than the MPI library regular entry points in the linking chain. Different names for both entry points prevent ambiguous or circular references at linking time.

This mechanism is a convenient option for source code users who want to instrument their programs for tuning or other purposes. However, the link-time instrumentation requires the object modules to be accesible to non-owners of the application. This is at odds with our black-box analysis goal.

Preloading

The Linux dynamic loader and linker provides a convenient mechanism for user intervention in the process of load and reference resolution, by means of environment variables in the context of the process requiring dynamic linking service. When spawning a new process, the user can point the environment variable LD_PRELOAD to shared objects to be scanned and relocated before any other libraries at load time of the executable. In this way, any pending references in the application object file are resolved against the user-provided preloaded, shared object, instead of any default system shared libraries.

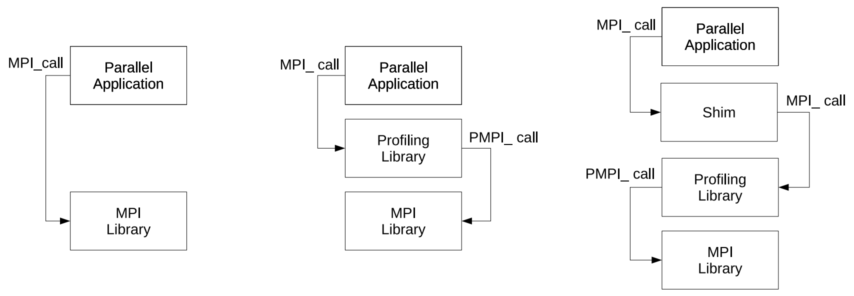
As it can be seen, preloading avoids explicit relinking operations or the need for source code or individual object files. We can exploit dynamic preloading for our profiling purposes in several ways:

1. By preloading an object-level profiling library, at execution time of arbitrary applications.
2. By writing our own profiling libraries, and run-time instrumenting arbitrary applications in suitable conditions.
3. By interposing our own functions contained in a shared object, between an application and a third-party profiling library which also exhibits suitable conditions.

Case 1 relieves us from the link-time requirement. Case 2 enables us to do our own profiling to closed applications, under a black-box approach. Case 3 is the case for shared libraries which exploit MPI's Profiling Layer Interface for performance analysis, because they expose MPI-prefixed symbols we can chain to. The technique in case 3 has the added value of leaving room for any further profiling work we desire to do inside our own functions, and yield all results in the same run.

A user shared object such as described in Case 3, interposed between application and library, is frequently called a shim, and may allow a programmer to intercept function calls in a transparent way, much resembling the design rationale of the MPI Profiling Interface Layer, only at run-time instead of link-time. The shim preloading idea has been used to do run-time instrumentation [23,15]. However, we have not found in literature the dynamic loading API and techniques considered as an alternative to link-time instrumentation.

The conditions for a binary application to be instrumented in this way are that the binary uses shared object, dynamic link libraries, and that those symbols corresponding to functions to be instrumented remain in UNKNOWN state to the linker (i.e. pending of relocation) at load time. This is the case for most binary executables using MPI which a user may be interested in tracing or profiling.



(a) Normal MPI call mechanism. (b) Interposing a PMPI-compliant profiling library. (c) Interposing a user shim.

Fig. 1: MPI Profiling Interface Layer.

Mixed instrumentation

One of our first needs in the way to characterizing applications is a convenient tool for inspecting the distribution of communication primitives. The scalable mpiP profiling library is a suitable tool for this purpose. However, mpiP is built upon the MPI Profiling Layer Interface, which means that relinking is needed, thus countering our black-box approach.

Using the shim technique, we have a way to do run-time instrumentation with our own code along with mpiP. As an MPI binary executable is normally an MPI shared library user which has pending references to MPI functions, we can interpose (by use of LD_PRELOAD) our own shim's MPI-prefixed symbols in the load path. Then, from within our instrumented MPI functions, we call their homonyms in the mpiP shared object library. The mpiP library functions then do their usual work for profiling purposes, finally calling PMPI-prefixed secondary entry points in the MPI library. The call path is as depicted in Figure 1c. We call this technique “mixed instrumentation”, because the user can combine her own source-level instrumentation with object-level instrumentation libraries.

As an example use case of mixed instrumentation, we were able to analyze process-to-process communication, building a bandwidth matrix, and then validate our results against mpiP output from the same runs. Another natural use case appears while analyzing performance of multi-level parallel applications on a hybrid cluster. When an application is developed using both shared-memory and message-passing programming techniques, we will often desire to take measurements related to both classes of events –at the same time; for instance, hardware events interspersed between message-passing function calls.

Laboratory

To test the overhead imposed by mixed instrumentation, we designed an experiment in two scenarios: a) Parallel executable, instrumented with mpiP library at the object level, and b) Same program, non-instrumented executable, but run-time instrumented via a no-op shim against mpiP functions. In Scenario a), calls to MPI functions in the executable are bound to mpiP wrapper functions, as in Fig. 1b. In Scenario b), calls to MPI in the executable are dynamically bound to no-op functions in our shim and then control falls into mpiP functions, as in Fig. 1c. While Scenario a) is the regular, suggested practice, b) is an example of the black-box style, mixed instrumentation, we intend to perform.

For the tests, we selected IS (integer sort), a program in NAS NPB3.3 benchmark suite [8]. This is the most communication demanding program in the suite. The IS program calls most of MPI communication primitives (MPI_Send, MPI_Irecv, MPI_Reduce, MPI_Allreduce, MPI_Alltoall, and MPI_Alltoallv). To take overhead measurements, we modified IS at source code level to record the total time to perform MPI calls in one run. We employed the same MPI wall clock time function (MPI_Wtime) as IS uses to compute the benchmarks results. The resolution of this timer can be queried by the function MPI_Wtick() which returns the number of seconds between successive clock ticks. In our system, the claimed resolution was 10^{-6}.

For every data size A, B and C (“classes” in NAS NPB parlance), two IS executables were compiled: an object-code instrumented program with mpiP library (for the “object-level” instrumentation scenario), and a non-instrumented one (for the “runtime-level” instrumentation scenario). We ran IS jobs for both scenarios in all data sizes (A, B and C) for a varying number of MPI processes (1..8), on a two-node cluster of dual-processor, dual-core machines (i.e. each machine hosts 4 compute cores) running 64bit Linux. We captured time in seconds for each run. One hundred runs of the whole test were recorded. Trimmed means were taken at several thresholds to get average $t_{object-level}$ and $t_{runtime-level}$ running time estimates. Finally, we computed the overhead estimate for every case as follows.

$$overhead = \frac{t_{runtime-level} - t_{object-level}}{t_{object-level}}$$

As a result, we found no significant overhead impact for any trimming threshold value. In fact, measurements may lead to think that preloading and interposition has a beneficial effect on performance in most cases, as suggested by the negative values. As depicted in Figure 2 (25% trimmed-means), the difference in execution time, relative to the "object-level" instrumentation, is in the order of 1%. The estimated overhead per MPI call is in the order of 0.01%, i.e. small enough to make the mixed instrumentation technique valuable for practical use.

	A	B	C
1	0.0004	0.0000	-0.0001
2	-0.0162	-0.0020	-0.0249
4	0.0007	0.0094	0.0178
8	0.0019	-0.0013	-0.0003

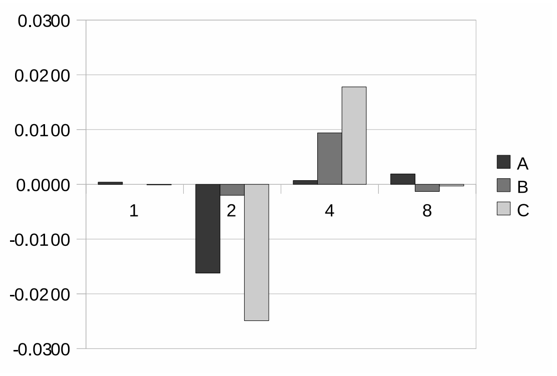


Fig. 2: Overhead for shim instrumentation, relative to object-level instrumented running time.

Conclusions and Future Work

The measured overhead imposes no practical problems to the idea of black-box profiling using this method. However, our experimental results warrant further research into the causes for lesser running times in Scenario b (i.e. when the shim technique is applied). In order to factor out systematic error, we conducted several other experiments over different hardware, reaching the same data patterns. Explanations related to the operating system environment will be considered as part of our future work.

The applicability conditions of the shim technique are limited by different features in profiling or tracing libraries. The MPE library, for instance, is unreductible "as is" because, as far as we know, the MPE software package provides no option to compile a shared object library. MPE-instrumented binaries are statically linked. VampireTrace, on the other hand, is a good candidate to apply our technique.

We conclude that dynamic preloading is a useful technique for run-time instrumentation of closed applications with performance analysis libraries which were originally designed for link-time instrumentation. While link-time profiling libraries can, in certain conditions, be used at run-time simply by preloading, mixed instrumentation adds the value of enabling us to interpose our own code in the same test run. This method can be used in a hybrid cluster environment to take profiling measurements related to events of other kinds.

To complete our tool, we are currently developing a second set of shim functions to intercept FORTRAN calls. Using our black-box profiling technique, we expect to instrument closed applications to know about their behaviour, and so draw models to predict performance in candidate running environments.

References

1. About OProfile. http://oprofile.sourceforge.net/about/.
2. Cómputo de altas prestaciones. http://hpc.uncoma.edu.ar/.
3. GNU gprof - table of contents. http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html.
4. Measuring and improving application performance with PerfSuite. http://perfsuite.ncsa.uiuc.edu/publications/LJ135/t1.html.
5. MPE. http://www.mcs.anl.gov/research/projects/mpi/www/www4/MPE.html.
6. MPI documents. http://www.mpi-forum.org/docs/docs.html.
7. mpiP: lightweight, scalable MPI profiling. http://mpip.sourceforge.net/.
8. NASA advanced supercomputing (NAS) division home page. http://www.nas.nasa.gov/.
9. Perf wiki. https://perf.wiki.kernel.org/index.php/Main_Page.
10. Performance analysis tools. https://computing.llnl.gov/tutorials/performance_tools/.
11. SourceForge.net: linux performance counters driver - project web hosting - open source software. http://perfctr.sourceforge.net/.
12. TAU - tuning and analysis utilities -. http://www.cs.uoregon.edu/research/tau/home.php.
13. TUD - ZIH - open trace format (OTF). http://tu-dresden.de.
14. Vampir. http://www.vampir.eu/.
15. K. J Barker, K. Davis, A. Hoisie, D. J Kerbyson, M. Lang, S. Pakin, and J. C Sancho. A performance evaluation of the nehalem quad-core processor for scientific computing. *Parallel Processing Letters*, 18(4):453–469, 2008.
16. Anthony Chan, William Gropp, and Ewing Lusk. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Sci. Program.*, 16(2-3):155–165, 2008.
17. Michael Collette. High performance tools and technologies. Technical Report UCRL-TR-209289, LLNL, 2004.
18. J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI for hardware performance monitoring on linux systems. In *Conference on Linux Clusters: The HPC Revolution*, 2001.
19. Zoltán Juhász. *Distributed & Parallel Systems - Cluster & Grid computing*. Springer, 2005.
20. Andreas Knüpfer. OTF tools.
21. A. Leko, H. Sherburne, H. Su, B. Golden, and A. D George. *Practical experiences with modern parallel performance analysis tools: An evaluation*. Citeseer.
22. Shirley Moore, David Cronk, Kevin London, and Jack Dongarra. Review of performance analysis tools for mpi parallel programs. In *In Y. Cotronis and J. Dongarra, Eds., Recent Advances in Parallel Virtual Machine and Message Passing Interface, 8th European PVM/MPI Users Group Meeting, Proceedings, LNCS 2131*. Springer Verlag, 1998.

23. R. Rabenseifner. Automatic profiling of MPI applications with hardware performance counters. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 22–22, 1999.
24. J. S Vetter and M. O McCracken. Statistical scalability analysis of communication operations in distributed applications. *ACM SIGPLAN Notices*, 36(7):123–132, 2001.