A Flexible and Dynamic Infrastructure for MPI Tool Interoperability

Martin Schulz and Bronis R. de Supinski
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
{schulzm,bronis}@llnl.gov

Abstract

The MPI standard provides tool builders with an efficient profiling interface, PMPI. Although many tools have successfully used this interface, it has three major drawbacks: a need to relink the application in order to use a tool; an inability to combine existing tools easily; and a lack of support for tool modularity. These limitations restrict tool flexibility and increase the threshold for using MPI tools.

We present P^{N}MPI , an infrastructure to load MPI tools dynamically and to chain multiple MPI tools for concurrent use. It works with existing PMPI tools, which can be transparently converted in binary form into loadable P^{N}MPI modules, and newly developed tools, which can exploit additional P^{N}MPI inter-tool communication services. We show that our implementation achieves our design goals, including ease-of-use and minimal overhead.

1 Motivation

Efficient and flexible support for tools is essential for any parallel programming environment. The Message Passing Interface (MPI) [4] provides an efficient mechanism to support profiling and tracing tools through its name shifted profiling interface (PMPI). Tool writers can transparently intercept any MPI call and wrap their own functionality around the actual MPI library invocation. Many popular tools use PMPI, including profilers like mpiP [8, 10], trace libraries such as those from Oregon [1] or Intel [11], or MPI correctness checkers like Umpire [9] or MARMOT [3].

Despite PMPI being widely used, it lacks flexibility. One problem is that it requires users to relink their application with the PMPI-based tool to include its functionality. More importantly, the PMPI interface only supports one tool at a time. The user cannot use multiple tools in a single run of their application, a significant limitation since not only application programmers might want to perform multiple performance analyses concurrently, but also tool builders cannot use existing tools, such as an MPI profiler, to evaluate the quality of the implementation of new tools, such as a correctness checker. Further, tool builders cannot easily create tool modules since functionality cannot be split into individual PMPI-accessing layers that could be reused by other tools. Thus, it discourages code reuse during tool development. Figure 1 (left) illustrates these points further. Applying multiple tools to an application requires linking the application with each tool individually to create multiple, separately run executables.

We present an infrastructure, P^{N}MPI , that eliminates these limitations of PMPI as the middle of Figure 1 shows. P^{N}MPI allows users to load dynamically and to execute (i.e., without relinking) one or more existing or newly developed PMPI-based tools (or modules) concurrently. We accomplish this by linking the P^{N}MPI infrastructure into applications by default. P^{N}MPI then transforms the wrapper libraries included in the PMPI tools into a single tool stack. Once initialized, P^{N}MPI redirects any MPI routine executed by the application into this dynamically created stack and independently calls each tool that contains a wrapper for the routine. This eliminates the need to create a separate executable for each tool and to run each tool separately.

Since P^{N}MPI is lightweight by design, it can be included in the default build process thereby removing the need for recompilation to include or remove a tool. P^{N}MPI also provides tool interaction functionality through services in the P^{N}MPI core. Thus, separate modules can now implement common tool functionality to improve code reuse, modularity, and flexibility as well as tool interoperability.

Figure 1 (right) shows just some of the possible usage scenarios for the P^{N}MPI infrastructure. Tracing and profiling tools can be combined transparently. It allows efficient MPI debugging by combining deterministic reply mechanisms with MPI checker libraries like Umpire [9]. A generic piggyback mechanism can be used as the basis of application level checkpointing mechanisms [7] or critical path detection [6]. The flexibility of the P^{N}MPI infrastructure supports many more scenarios.

[0]This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48 (UCRL-CONF-221608).

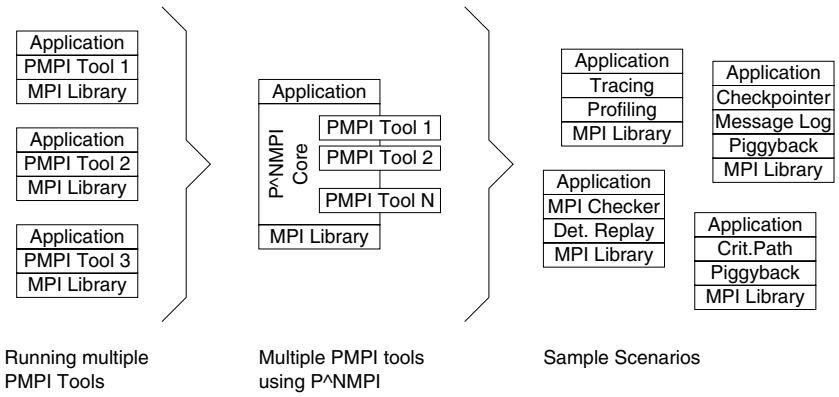

Figure 1. Multiple tools without (left) or by chaining them together with P^{N}MPI (middle); sample scenarios for P^{N}MPI (right).

2 MPI Tools

2.1 The MPI Profiling Interface

MPI includes a portable and efficient mechanism for tool development, the PMPI Profiling Interface. Tools can create wrappers for any MPI routine and then transparently insert them between the MPI library and the application. PMPI, a name-shifted interface, makes all MPI functionality available through alternate names (with the prefix *PMPI_* instead of *MPI_*). Tools can overwrite the existing MPI functions, add their own processing, and then invoke the original MPI functionality using the shifted names.

Figure 2 shows the common PMPI implementation through weak symbols. All original MPI routines are declared using weak symbols. Thus, any library linked to the application that uses the same names will overwrite them. The PMPI names, on the other hand, are regular symbols; tool wrappers can use them for the actual communication library work. Thus, tools can transparently register any MPI invocation and record its arguments by wrapping the corresponding MPI routine. Many MPI tools use this infrastructure, including profilers [8, 10], tracers [1, 11] and correctness checkers [9, 3].

2.2 Limitations

We address three major shortcomings of PMPI:

Restricted to one tool at a time:

The PMPI interface allows only one tool to be linked between the MPI library and the application at a time. Users must link and run with each tool separately in order to apply multiple PMPI tools. Further, combining results from multiple runs requires a highly deterministic application execution, a property that does not hold for many codes or systems. Even worse, application of one tool to another (e.g., evaluation of the performance impact of a correctness tool) is not supported.

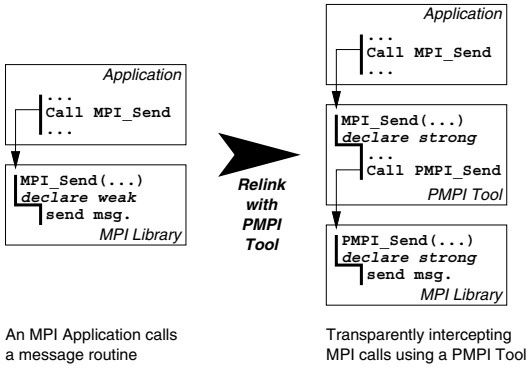

Figure 2. Typical implementation of the PMPI library using weak symbols for all MPI routines.

Need to relink for each tool:

In order to use a PMPI tool, the application must be relinked to include the PMPI wrappers. While much less invasive than requiring recompilation, it still requires tool users to have all application binary components. Also, specific MPI tools must be integrated into the make environment or build process, which can be difficult in larger projects.

Limited support for tool modularity:

Many MPI tools require similar components, such as logging, handle replacement, request tracking, or piggyback messaging. Generic tool libraries must provide this common functionality since PMPI does not include it. However, these libraries cannot take the form of PMPI tools themselves, which heavily restricts their design. Thus, tool builders often must rewrite such components from scratch.

In summary, these shortcomings lead to a restricted tool environment and raise the bar for users to apply tools in their MPI applications. They also greatly limit or even eliminate any interaction between tools. Finally, they increase tool development efforts by discouraging code reuse.

3 P^{N}MPI : Chaining MPI Tools

Our novel P^{N}MPI infrastructure overcomes PMPI's limitations while maintaining its advantages and leveraging the existing code base. P^{N}MPI can dynamically load and chain multiple PMPI tools into a single tool stack and then interject this complete stack between the target application and the library without changing the view for each individual tool. It enables the user to combine arbitrary MPI tools and the tool developer to rely on services provided by other tools without having to reimplement them. Our P^{N}MPI tool infrastructure meets these requirements:

1. Concurrently execute multiple PMPI tools during a single application execution.
2. Work with any existing MPI implementation.
3. Transparent adaptation to the underlying MPI layer.
4. Negligible overhead when not activated.
5. Dynamically load modules (i.e., no relinking).
6. Low overhead for actually intercepted MPI routines.
7. Binary backward compatibility with existing tools (i.e., run any PMPI-based tool without recompiling it).
8. Optional communication and interoperation of newly developed tools, including information exchange.

3.1 P^{N}MPI Architecture

Once P^{N}MPI is included into the application, users can dynamically specify a set of PMPI tools to apply in a single application execution (Requirement 1). These individual tools are dynamically loaded at runtime and integrated into a single tool chain that is inserted between the application and the MPI library.

P^{N}MPI consists of three main components (Figure 3): a stub library, itself written in PMPI (Requirement 2), that integrates the complete tool chain into the MPI application; the P^{N}MPI core managing and controlling the tool chain; and a configuration and loader module to specify the PMPI modules to use and to initialize the infrastructure. In addition, it also includes an automatic customization and adaptation mechanism that retargets P^{N}MPI to the underlying MPI implementation on the target platform, as well as a transparent migration tool to integrate existing binary PMPI tools into P^{N}MPI .

The stub library uses PMPI to cover all MPI calls that the native MPI layer provides, which can vary between MPI implementations and does vary between MPI-1 and MPI-2. Thus, the P^{N}MPI installation process generates this library automatically (Requirement 3). During this process, the stub library generator parses the native header file and extracts all routines with a PMPI implementation along with their parameters. It then generates corresponding stub routines that direct the calls into the P^{N}MPI core if and only if at least one of the PMPI tools used in the current session implements this particular routine within its wrapper layer. If no active tool implements a routine, the stub wrapper directly hands the call to the underlying MPI library without further processing (Requirement 4).

This shortcut mechanism is introduced to keep the P^{N}MPI layer lightweight and to ensure that it incurs virtually no overhead for any MPI call not intercepted by a PMPI tool. As a consequence, an application's regular build process can include linking in the P^{N}MPI stub library into the program by default without concern for additional overhead. This can even be done site-wide transparently to the user, e.g., by integrating it into the *mpicc* compiler script. In either case, user's can then add PMPI tools to any execution, irrespective of whether they intended to do so during the build process (Requirement 5). This is especially helpful for debugging purposes where it is often unknown a priori what problems to expect.

If a routine is included in at least one tool, the stub library passes control to the P^{N}MPI core, which calls all tool layers that implement a wrapper for this routine. For this purpose, P^{N}MPI maintains separate link stacks for each MPI routine to avoid searching the entire tool stack for each activation (Requirement 6). The initialization process extracts the starting addresses of all MPI routines in each PMPI tool and populates these link stacks.

During the execution of a wrapper routine, any call to an PMPI routine must lead back to the P^{N}MPI core rather than to the MPI implementation to enable P^{N}MPI to gain control between layers and invoke the next one. This mechanisms works in scenarios where a MPI wrapper only calls its direct PMPI counterpart or where it uses one or more other PMPI calls to accomplish its task.

To achieve this redirection of PMPI calls, P^{N}MPI includes an external patch utility that transforms PMPI tool binaries into P^{N}MPI modules. This tool scans the dynamic symbol table and rewrites the names of PMPI routines to match internal P^{N}MPI core routine names. During the load process of the module, the calls to the PMPI routines are linked to the core rather than the MPI library, ensuring that the core gains control after each tool invocation. This patch utility directly modifies existing tools for use with P^{N}MPI without recompiling them (Requirement 7).

3.2 Configuration Mechanism

Once the P^{N}MPI stub library is linked into the application, the user can dynamically specify a PMPI tool set to load into the framework as *PMPI modules* (Requirement 1). The tool set can include existing ones and those specifically developed for P^{N}MPI . The former can be transparently transformed using the patch tool (Requirement 7).

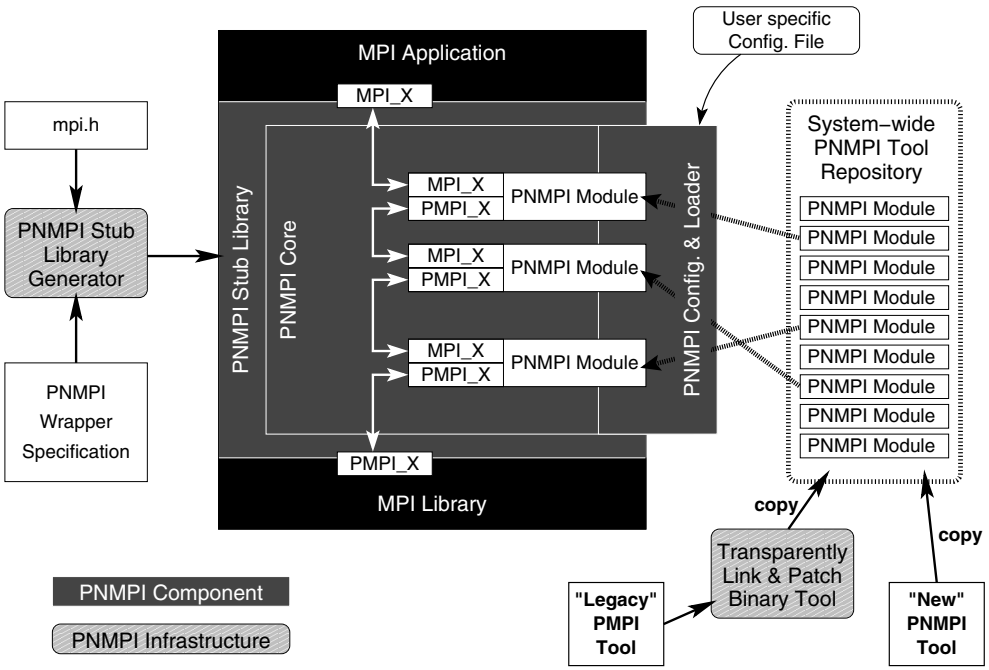

Figure 3. P^{N}MPI components and their interactions.

A configuration file (in the user's home directory or specified by an environment variable) controls the load process. This file lists the PMPI modules to load during P^{N}MPI initialization. The core's *MPI_Init* wrapper loads, instantiates and chains these modules in the file-specified order.

Thus, users can specify new tools for each run without changing the make or link process or the binary itself; they only change the configuration file. Further, PMPI tools can be removed after the respective analysis is completed, while the same binary can continue to be used for production runs. Hence, the application no longer has to be relinked to add, change, or remove tools (Requirement 5).

3.3 Inter-Tool Communication

Running multiple tools at the same time also opens the possibility to develop cooperative tools. This promotes both tool modularity and code reuse by allowing tools to by layered on top of functionality provided by other tools.

To enable this functionality the P^{N}MPI core provides a set of services to implement inter-tool communication (Requirement 8). The core includes a publisher/subscriber interface that tools can use to register services they offer and to search for services offered by other tools. A detailed discussion of these services is beyond the scope of this paper.

4 Experimental Setup

We conducted all experiments on *MCR*, a 1152 node, Quadrics's QsNet (Elan-3) cluster at LLNL. Each node has 2.4 MHz Dual Xeon processors and 4 GByte memory. The system uses the CHAOS–3.1 Linux distribution, which is based on Red Hat Enterprise Linux 4. All codes were compiled with gcc 3.4.4 and linked to Quadrics's MPI implementation. P^{N}MPI was automatically customized for Quadric's MPI during its installation.

Besides the microbenchmarks described below, we use two well-known scientific applications: SMG2000 and HPL. The former is a Semicoarsining Multigrid Solver based on the hypre library [2]. The latter is a portable high-performance implementation of the Linpack benchmark [5]. We executed these on four nodes/eight CPUs, 32 nodes/64 CPUs, and 256 nodes/512 CPUs. The working set size for SMG2000 was a 60x60x60 cube per node and we ran HPL with N=10000 on eight CPUs, N=14000 on 64 CPUs, and N=20000 on 512 CPUs.

Most experiments[1] were run at least five times each and the numbers presented are the minimal values[2] of all experiments, since these runs are likely to show the least external perturbation. Nevertheless, all numbers below are influenced by unavoidable system noise, which from previous experience can be several percent for individual runs.

[1] Experiments on 512 CPUs were run less frequently due to resource limitations. The noise on this data is therefore expected to be higher.

[2] Except for bandwidth numbers where we chose the maximal value.

5 P^{N}MPI Overhead

5.1 Overhead model

The P^{N}MPI performance overhead for each MPI call can be split into three components: one-time overhead in the stub library α_{stub}, one-time overhead in the core for any routine activation α_{core}, and the cost per tool. The latter consists of the cost in P^{N}MPI to run through the tool chain β and the cost of the tool itself. The following formula describes the complete model for the overall overhead.

$$T_{overhead} = \alpha_{stub} + \alpha_{core} + \sum_{i=1}^{N_{tools}} (\beta + T_{tool}(i))$$

In order to model the overhead of P^{N}MPI only, we ignore the overhead of each tool in this simplified formula:

$$T_{overhead} = \alpha_{stub} + \alpha_{core} + \beta * N_{tools}$$

Based on this model, we expect overhead linear in the number of tools loaded into P^{N}MPI plus a constant.

5.2 Model Verification

To verify this model, we first measure both bandwidth and latency for native execution and execution with the P^{N}MPI stub library included and compute the difference (α_{stub}). Table 1 shows that latency increases minimally (66ns) and bandwidth is essentially unchanged.

Once the application includes the stub library, we can measure the impact of adding tools to P^{N}MPI . For this we use an empty PMPI wrapper layer ($T_{tool}(i) = 0$), which simply intercepts all MPI routines and immediately calls the matching PMPI routines without any further processing. Figure 4 shows the message latency for the combined execution of zero to 100 of such empty layers[3]. As predicted by the model, the overhead increases linearly with the number of included tool layers at an average overhead of 50ns for each tool layer (β). Also, these experiments show that α_{core} is negligible for all practical purposes.

[3] Although users are unlikely to apply even 100 tools concurrently, we test P^{N}MPI with up to 1000 concurrently loaded tools to evaluate its robustness. The measured overheads continued to fit the model.

Benchmark	Native	P^{N}MPI	Overhead
Latency	4.760μs	4.824μs	0.066μs
Bandwidth	304.66 MB/s	304.62 MB/s	0.04 MB/s
SMG, 8 CPUs	63.15s	63.29s	0.22%
SMG, 64 CPUs	77.19s	77.39s	0.26%
SMG, 512 CPUs	100.91s	100.472s	-0.44%
HPL, 8 CPUs	79.17s	79.93s	0.95%
HPL, 64 CPUs	30.62s	30.89s	0.88%
HPL, 512 CPUs	17.84s	17.82s	-0.11%

Table 1. Overhead α_{stub} of the P^{N}MPI stub library.

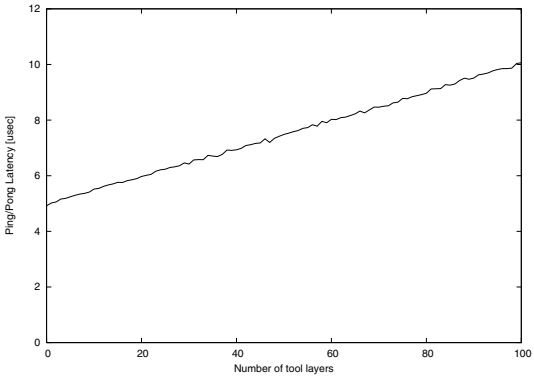

Figure 4. Impact on message latency for varying number of empty layers.

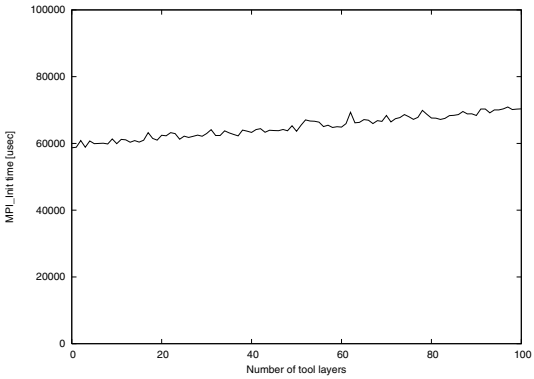

Figure 5. Time for *MPI_Init* for varying number of empty layers.

5.3 Module Initialization

We also measured the overhead on *MPI_Init* of P^{N}MPI tool module loading. Figure 5 shows this overhead also increases nearly linear with the number of tools, but still stays within 12ms or 20% compared to the native *MPI_Init* for 100 tools. This cost is certainly acceptable since it is a one time overhead.

5.4 Delay Microbenchmark

To illustrate the impact of multiple tools on the performance and to demonstrate the functionality and correctness of P^{N}MPI , we generated a PMPI benchmark layer that introduces an artificial delay of 100μs into every send call and acts like the empty layer for each other call. Figure 6 shows the latency with zero to ten of these layers activated. As expected, each tool layer works independently and adds 100μs to the overall latency measured at the application level.

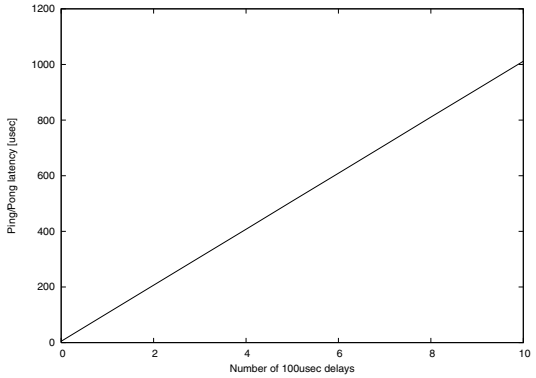

Figure 6. Impact on message latency for varying number of layers introducing 100μs delay each.

5.5 Application Overhead

Next we applied the same overhead experiments to both SMG2000 and HPL to measure how the overhead seen in individual routines translates to application overhead. The results in Table 1 shows that the stub library infers no or only minimal overhead. Especially at large scale, other performance effects dominate completely masking the influence of the P^{N}MPI stub library.

With increasing numbers of empty tool layers the overhead remains low (Figure 7). SMG2000 is generally more affected than HPL, which can be explained by looking at the applications' internal structures: HPL has less communication requirements than SMG2000 and hence is less impacted by increased latency. Only at larger scale and with more than 20 tools, HPL shows some performance degradation, most likely caused by HPL's synchronous structure, which is more susceptible to load imbalance caused by noise. Nevertheless, the overhead of about 1% for 20 tools and 2-5% for 60 tools is still extremely low.

5.6 Memory Overhead

Memory overhead is also a concern with an infrastructure such as P^{N}MPI , since any memory that it uses is no longer available to the application or other tools. In this section, we present a model for the memory consumption of P^{N}MPI .

We model P^{N}MPI 's memory usage with equations for global variables, heap allocated memory, and stack space utilized during MPI invocations. These equations depend on two parameters: N, the number of modules loaded, and M, the number of MPI routines that the PMPI interface implements. We assume a 32 bit architecture, as is the case on *MCR*. On 64 bit architectures the memory footprint will be larger at most by a factor of two, mainly due to the larger pointer size.

P^{N}MPI uses a static pointer array with an entry for each MPI routine accessible through the PMPI interface. This array stores the individual function stacks. Further, P^{N}MPI maintains a bit vector in its stub library to limit performance overhead for routines that no tool intercepts. With an additional 24 bytes for global configuration information, the following models global memory usage:

$$M_{global} = 24 + 4 * M + \lceil M/8 \rceil \ \ (Bytes).$$

The heap is used for two data structures: a descriptor for each loaded module, which is currently 116 bytes, and the function stacks for each function used by at least one tool. The number of these stacks allocated is always smaller or equal to M; the following therefore represents the worst case behavior, i.e., the case in which every MPI function is wrapped by at least one tool:

$$M_{heap} \leq 116 * N + 4 * M * N \ \ (Bytes).$$

If any tool calls MPI routines recursively, we cannot bound memory usage. However, if calls in every tool are restricted to PMPI rotuines, P^{N}MPI will not use more than eight bytes for each invocation of its core. This includes the initial MPI call and one for each active PMPI tool:

$$M_{stack} = 8 * (N + 1) \ \ (Bytes).$$

In our experiments $M = 128$, so the total memory overhead is:

$$M_{overhead} \leq 560 + 636 * N \ \ (Bytes).$$

The above is the worst case: many MPI tools only wrap a few MPI routines. Since P^{N}MPI allocates only required function stacks, these tools incur reduced heap usage. We could further reduce memory overhead through a smaller limit on module names, which is currently set to a comfortable 100 characters.

6 Case Study: Combining Independent Tools for Profiling and Tracing

The most common MPI performance analysis tools generally use one of two basic mechanisms: profiling, i.e., the collection of statistical data summarizing the execution characteristics; and tracing, i.e., the collection of all MPI-level events throughout the execution of an application. In some situations, however, it is necessary to gather both tracing and profiling data from one application. Using the conventional PMPI profiling layer, both tools need to be each individually linked with the application and executed separately. Using P^{N}MPI , both tools can be executed together in a single application run.

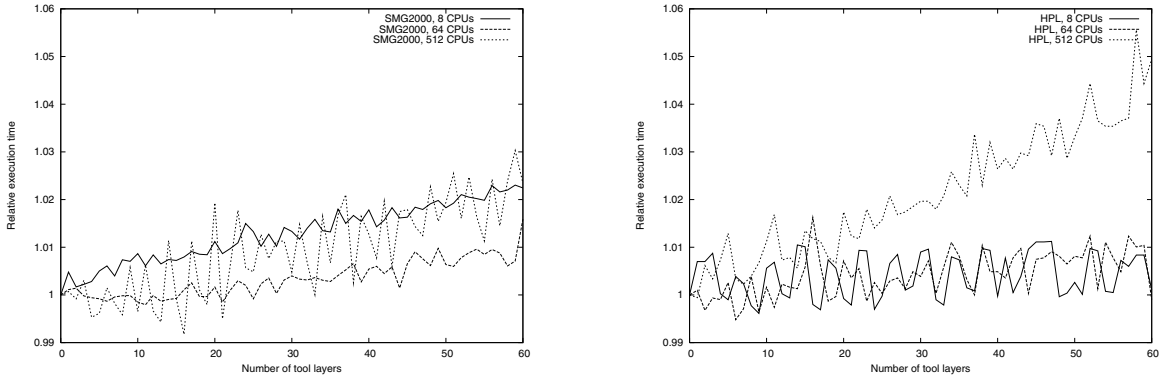

Figure 7. Overhead for applications when used with varying numbers of empty layers: Left: SMG2000, Right: HPL.

6.1 Setup and Configuration

In the following we use the mpiP [8, 10] profiler, developed and maintained at LLNL, and the MPI trace wrapper library from the TAU tool suite [1] developed and maintained by the University of Oregon. First, we convert both tools into PMPI tool modules by linking them into two separate shared objects and applying the P^{N}MPI patch utility to modify the dynamic symbol tables. We then store the completed modules in the P^{N}MPI tool module library, which is specified by the *PNMPI_LIB_PATH* environment variable.

To activate both modules, the application must be linked with the P^{N}MPI stub library and the modules must be specified in the configuration file *.pnmpi-conf* stored either in the current working directory or in the user's home. Figure 8 shows the file used to combine the tracing module, named *tautrace*, and the profiler, named *libmpip*. The remaining lines in the file (starting with #) are comments.

6.2 Performance Results

We tested this combination of tools using the experimental setup and the application benchmarks introduced in Section 4: SMG2000 and HPL. Table 2 shows the results of the experiments without P^{N}MPI , with each tool loaded statically, with each tool loaded individually using P^{N}MPI , and both tools loaded concurrently.

Note that in general the largest scale runs exhibit more noise and due to the tools' higher I/O requirements, especially in the tracing library, also lead to larger overheads. Further, since the I/O system is shared with other cluster partitions, performance results can be highly variable, even with multiple runs per data point.

```
#Combine MPI tracing and profiling

#tracer first
module tautrace

#profiler second
module libmpip
```

Figure 8. Configuration file to combine tracing and profiling (comment lines are denoted with #).

As in the previous section, the overhead of the P^{N}MPI layer alone is minimal. Similarly, the difference between using a single tool using PMPI and P^{N}MPI is within the noise level of the experiments. The tracing tool incurs significantly more overhead than the profiler and the overhead increases with growing processor numbers. This is expected, since a tracer must individually inspect and record each MPI event, while a profiler only maintains aggregate metrics.

The combined tool overhead using P^{N}MPI is generally comparable to the sum of the overhead caused by running each tool individually. However, some overhead is added by the second tool in the chain handling not only MPI traffic caused by the application, but also with additional traffic due to the first tool using additional PMPI calls for its functionality.

In summary, we find that P^{N}MPI overhead is negligible compared to the overhead caused by the actual tools. However, combining tools for concurrent use can lead to accumulative overheads from the individual tools. Users have to be aware of this additional source of perturbation, especially when using P^{N}MPI for performance analysis.

7 Conclusions

PMPI is a successful standardized interface that supports a wide array of parallel tools, from MPI profilers and tracers to MPI correctness checkers. However, users must relink in order to include a tool, requiring a change to the build process for each tool employed, and users can only use one tool at a time. The latter means that users must generate a separate executable and rerun the application for each tool. Further, tools cannot easily be built by composition and one tool cannot be applied to another.

	Baseline runtime [sec]	Overhead					
		static mpiP	static tracer	P^NMPI no tools	dynamic mpiP	dynamic tracer	mpiP and tracer
SMG2000, 8 CPUs	63.15	2.30%	7.10%	0.22%	2.50%	7.56%	9.49%
SMG2000, 64 CPUs	77.19	1.30%	21.01%	0.26%	1.55%	21.72%	25.31%
SMG2000, 512 CPUs	100.91	3.69%	96.90%	-0.44%	3.00%	98.68%	101.23%
HPL, 8 CPUs	79.17	1.64%	1.98%	0.95%	1.86%	2.81%	3.17%
HPL, 64 CPUs	30.56	1.57%	15.45%	0.16%	1.87%	13.45%	12.50%
HPL, 512 CPUs	17.84	12.33%	4271.69%	-0.11%	4.71%	3323.93%	2068.5%

Table 2. Overhead results for running mpiP and/or the TAU tracer module with and without P^NMPI .

We presented an infrastructure, P^NMPI , that builds on the advantages of PMPI to add these missing capabilities. P^NMPI allows users to load and activate arbitrary numbers of PMPI tools dynamically by simply editing a configuration file without the need for recompilation or relinking. This provides the ability to add tool support when necessary, even if not considered at the application's build time, and to remove tools when the necessary analysis steps are completed, using the same binary.

P^NMPI maintains a repository with all tool modules available for use within MPI applications. Besides tools with explicit P^NMPI support, a binary patch tool can modify existing PMPI tools without recompiling so the repository can include them. This opens P^NMPI to third-party tools to which users often don't have source access.

P^NMPI provides low overhead for microbenchmarks and large scale applications and can run hundreds of tools at the same time. Our case study used P^NMPI to combine existing MPI tools for tracing and profiling in a single run with minimal additional overhead.

Overall, P^NMPI eases the use of MPI tools, by allowing them to be dynamically linked, and provides efficient interoperability between multiple tools. This interoperation can be cooperative or transparent by concurrently executing multiple tools independently. P^NMPI supports tool layering, which leads to more modularity and code reuse in tool design. In the future we will continue to investigate these inter-tool cooperation issues and extend the model for a more general approach, including the ability to not only stack MPI modules, but to allow users to specify a context sensitive dataflow between multiple modules. This enables the use of PMPI tools, e.g., on only subsets of MPI routines or time slices of applications and thereby further increases the flexibility in applying tools for MPI applications.

References

[1] R. Bell, A. Malony, and S. Shende. ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pages 17–26, Aug. 2003.

[2] R. Falgout and U. Yang. hypre: a Library of High Performance Preconditioners. In *Proceedings of the International Conference on Computational Science (ICCS), Part III, LNCS vol. 2331*, pages 632–641, Apr. 2002.

[3] B. Krammer, M. Müller, and M. Resch. Runtime Checking of MPI Applications with MARMOT. In *Mini-Symposium on Tools Support for Parallel Programming at ParCo 2005*, Sept. 2005.

[4] Message Passing Interface Forum (MPIF). MPI: A Message-Passing Interface Standard. Technical Report, University of Tennessee, Knoxville, June 1995. http://www.mpi-forum.org/.

[5] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. Hpl - a portable implementation of the high-performance linpack be nchmark for distributed-memory computers. Available at http://www.netlib.org/benchmark/hpl/.

[6] M. Schulz. Extracting Critical Path Graphs from MPI Applications. In *Proceedings of IEEE Cluster 2005*, Sept. 2005.

[7] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Implementation and Evaluation of a Scalable Application-level Checkpoint-Recovery Scheme for MPI Programs. In *Proceedings of IEEE/ACM Supercomputing '04*, Nov. 2004.

[8] J. Vetter and C. Chambreau. mpiP: Lightweight, Scalable MPI Profiling. http://www.llnl.gov/CASC/mpip/, Apr. 2005.

[9] J. Vetter and B. de Supinski. Dynamic software testing of mpi applications with umpire. In *Proceedings of IEEE/ACM Supercomputing '00*, Nov. 2000.

[10] J. Vetter and M. McCracken. Statistical Scalability Analysis of Communication Operations in Distributed Applications. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), 2001*, 2001.

[11] I. Website. Intel (R) Cluster Toolkit, Inte (R) Trace Analyzer and Collector 6.0. http://www.intel.com/cd/software/products/asmo-na/eng/cluster/clustertoolkit/244170.htm#trace, 2006.