P^{N}MPI Tools:
A Whole Lot Greater Than the Sum of Their Parts

Martin Schulz and Bronis R. de Supinski

Center for Applied Scientific Computing[*]
Lawrence Livermore National Laboratory
Livermore, CA 94551
{schulzm,bronis}@llnl.gov

ABSTRACT

P^{N}MPI extends the PMPI profiling interface to support multiple concurrent PMPI-based tools by enabling users to assemble tool stacks. We extend this basic concept to include new services for tool interoperability and to switch between tool stacks dynamically. This allows P^{N}MPI to support modules that virtualize MPI execution environments within an MPI job or that restrict the application of existing, unmodified tools to a dynamic subset of MPI calls or even call sites.

Further, we extend P^{N}MPI to platforms without dynamic linking, such as BlueGene/L, and we introduce an extended performance model along with experimental data from microbenchmarks to show that the performance overhead on any platform is negligible. More importantly, we provide significant new MPI tool components that are sufficient to compose interesting MPI tools. We present three detailed P^{N}MPI usage scenarios that demonstrate that it significantly simplifies the creation of application-specific tools.

1. MOTIVATION

The MPI specification includes a standardized interface for tools to wrap MPI calls and thereby to monitor the communication of applications: the name shifted PMPI interface. While it is has been successfully used for many tools, including mpiP [12], Umpire [13], or TAU [2] it comes with severe limitations: PMPI limits users to a single tool at a time, preventing tool modularity and interaction, and forces any tool to always be applied to all MPI ranks. The result are complex, monolithic tools with limited code reuse, which are hard to implement and to debug.

To overcome these deficiencies of PMPI, while maintaining its successful and widely used interface, we have introduced P^{N}MPI [11]. It enables users to load and to stack arbitrary PMPI tools dynamically and run them concurrently. In this work we extend on this basic infrastructure to enable the composition of novel types of tools that can exceed their original functionality. Our extensions include:

- A registration mechanism that enables tool service layers to extend the core framework;
- *Switch Modules*, a mechanism to select tool stacks dynamically;
- An MPI virtualization module that provides multiple, separate MPI environments.

These new capabilities can be used to compose new tools directly from existing tools or out of a library of generic services or to restrict tools transparently to relevant subsets of applications. Our registration mechanism enables code reuse by modularizing common tasks, like datatype flattening or request tracking, without having to recode them in every tool. *Switch Modules* support both external steering and the multiplexing of existing tools to dynamic subsets of MPI jobs. MPI virtualization allows the transparent coupling of cooperating, yet separate application components without requiring the recoding that might be needed under MPI-2. We illustrate the power of our extensions through three widely different case studies: using a cooperative tool stack for the generation of message checksums, applying multiple copies of an existing MPI profiling tool to different sets of communicators, and combining multiple SPMD applications into a single MPI job.

In addition to these new capabilities, we also present an alternative design of the P^{N}MPI core framework that allows its use on machines without dynamic linking support. As a result, P^{N}MPI now runs on machines such as Blue Gene/L and Red Storm, which operate with lightweight compute node kernels, as well as more standard clusters that use Linux or other Unix variants such as AIX. Thus, P^{N}MPI now represents a truly global solution for managing and composing MPI tools.

The remainder of this paper is organized as follows. In Section 2 we present the concepts behind P^{N}MPI followed in Section 3 with details on its implementation. Section 4 extends the existing P^{N}MPI performance model to reflect the new capabilities enabled by *Switch Modules* and evaluates it on a large scale Linux cluster. In Section 5 we illustrate the capabilities of P^{N}MPI with the three case studies introduced above followed by a short discussion. Section 6 briefly covers related work, followed by some final remarks in Section 7.

[*]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-229978).

2. COMPOSING TOOLS IN P^{N}MPI

P^{N}MPI eliminates the restriction of a single PMPI tool layer per execution [11]. In this work we generalize the basic concept of a dynamically defined and loaded tool stack managed by P^{N}MPI to dynamic MPI tool chains potentially consisting of multiple tool stacks. To manage these chains, the P^{N}MPI infrastructure provides a set of services that allow individual tools to interact with each other, to rely on cross-tool services, or to switch between tool stacks dynamically. This enables the composition of a new class of modular MPI tools previously only possible in complex, monolithic tool designs. In this section we describe the individual components and services provided by P^{N}MPI to create MPI tool chains as well as the composition process itself.

2.1 Building MPI Tool Chains

The basic component in any P^{N}MPI tool chain is a single PMPI tool, in the following referred to as a P^{N}MPI module. We distinguish between two types of modules: transparent and P^{N}MPI -aware. The former type relies only on the original PMPI interface to wrap MPI calls and includes all tools that can run on MPI applications without the P^{N}MPI infrastructure. The inclusion of this type of modules enables P^{N}MPI to support any existing MPI tool, even in binary form, and ensures that P^{N}MPI's capabilities are a strict superset of the PMPI interface. Alternatively, P^{N}MPI -aware modules have been designed to specifically work within the P^{N}MPI environment and therefore can take advantage of P^{N}MPI services beyond wrapping any MPI call to implement their functionality.

Users can combine any P^{N}MPI module, transparent or P^{N}MPI -aware, into tool stacks that recursively wrap all MPI calls included in the respective modules. Figure 1 (left) illustrates how two PMPI tools, which are designed to individually intercept MPI calls of applications, are composed into a single tool stack. With our extensions to P^{N}MPI, users can define multiple independent tool stacks to create a single overall composite tool. Special P^{N}MPI aware modules, which we call *Switch Modules*, support switching between stacks at runtime. For this, the P^{N}MPI infrastructure exports an extended PMPI interface, through which a target stack can be specified for each PMPI function call.

Figure 1 (right) shows an example of a tool chain using a switch module. Each MPI call from the application is first directed into *Stack A* before reaching the *Switch Module*. The *Switch Module* forwards the MPI call to either *Stack B* or *Stack C* based on a programmer specified selection criteria. At the end of either of these stacks, the MPI call is forwarded to the PMPI interface of the MPI library. On return from the MPI library, P^{N}MPI traverses the selected tool stacks in reverse order. The programmer of the *Switch Module* can use any data available to it, including global state or MPI function arguments, to select the appropriate target stack. Thus, P^{N}MPI supports external steering based on global state. Similarly, certain communicators, datatypes, or sender/receiver pairs can determine the selection.

Switch Module control can be transparent to the tools inside the tool stacks. Further, modules can be replicated in multiple stacks. Thus, *Switch Modules* enable transparent extensions of existing tools. For example, P^{N}MPI can apply a single tool to multiple independent subsets of MPI calls without having to modify the tool itself, which we show in Section 5.2 allows communicator-specific profiling.

2.2 Service Registration and Execution

P^{N}MPI -aware modules must register themselves with the P^{N}MPI infrastructure. Once registered, they can offer a set of services to other modules using a publish/subscribe interface. Modules can query for services, obtain a global function pointer, and then use this pointer to invoke cross module functionality. All services include a signature provided by the module offering the service that specifies the types of all arguments. Modules querying this service must provide a matching signature to avoid type conflicts.

Cross module services can encapsulate tasks commonly required by MPI tools. Typical examples include tracking of request objects or walking of MPI datatypes. With P^{N}MPI , tools that require these functionalities can request them from service modules that provide them. This enables efficient code reuse, promotes modularity, and enables quick prototyping of new tools.

3. P^{N}MPI IMPLEMENTATION

We have implemented P^{N}MPI on a wide range of platforms including Linux-based clusters using IA-32, IA-64, and x64_64 based CPUs, Power-based AIX clusters, and BlueGene/L. The implementation is highly portable: only details of the stacking mechanism are system specific.

3.1 Architecture

P^{N}MPI's base architecture, shown in Figure 2, consists of three major components: the P^{N}MPI stub library intercepts the application's MPI calls; the core component constructs and executes the tool stacks; and the configuration and loader initializes the tool stacks. The user specifies a configuration file that controls the configuration and loader, which must locate, load, and instantiate the specified P^{N}MPI modules and assemble them into logical stacks. The P^{N}MPI core then connects the logical stacks to a single tool chain using the stacking mechanism appropriate for the target architecture. It also provides the service registration and query mechanisms that allow modules to interact. The stub library, which is itself built using PMPI to intercept all MPI calls, activates the tool chain by forwarding all MPI invocations wrapped by at least one loaded tool to the P^{N}MPI core. The stub library directly forwards all other MPI invocations to the MPI library to minimize overhead.

The P^{N}MPI installer ensures complete coverage of all calls offered by the underlying MPI implementation through automatic generation of the MPI stub library. During the build process it parses the `mpi.h` header file, extracts all routines that have PMPI counterparts, and instantiates the corresponding routines in the stub library and the P^{N}MPI core.

All components of P^{N}MPI are written in C or C++. Fortran applications are supported through a wrapping layer inside the stub library. This layer transform MPI calls made in Fortran into the equivalent C calls before invoking the P^{N}MPI core. Other languages could be supported similarly, if needed.

3.2 Creating Tool Stacks

The implementation of the P^{N}MPI core, which creates and executes the tool stacks, depends on whether or not the target operating system supports dynamic linking. In either case, however, P^{N}MPI works with binary versions of PMPI tools: it does not require recompilation of existing tools.

Dynamic Stacking

If the host OS supports dynamic linking through shared libraries, the runtime P^{N}MPI initialization creates the tool stacks. This requires that all P^{N}MPI modules are available as shared libraries; if not, we convert them with automated support. We then use a binary rewriter to modify the dynamic symbol table so all PMPI calls are redirected to the P^{N}MPI core. Both steps also apply to existing binary tools, allowing integration of commercial tools without source code access.

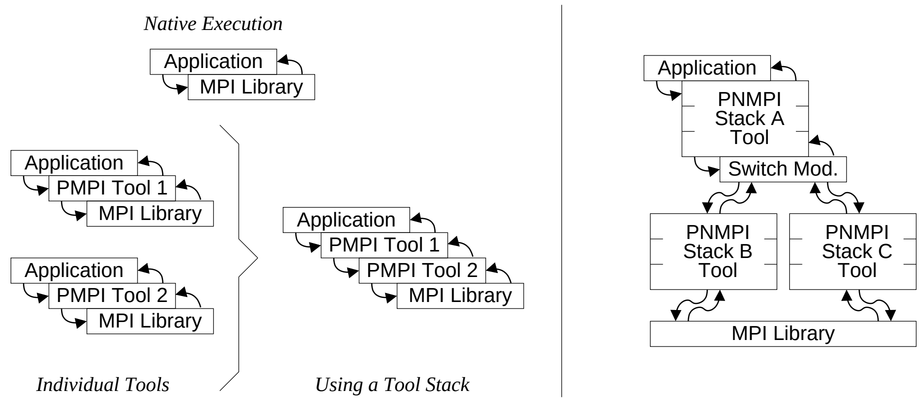

Figure 1: Combining multiple tool modules into a tool stack (left); dynamically choosing tools stacks (right).

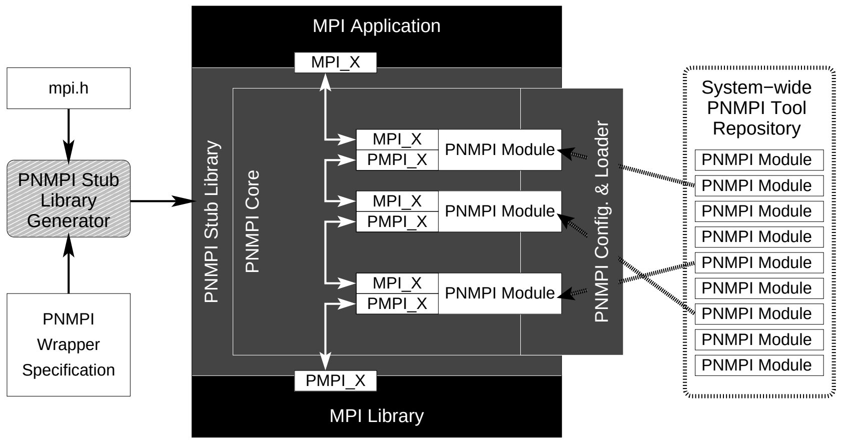

Figure 2: P^{N}MPI components and their interactions.

At application start P^{N}MPI parses the configuration file, loads the patched shared libraries, and searches them for wrapped MPI functions. The P^{N}MPI core uses this information to create separate wrapper stacks for each MPI routine. When the application calls an MPI routine, the stub library forwards it to the core, which then calls the wrapper function of the appropriate tool layer. The wrapper itself calls the matching PMPI routine, which returns control to the P^{N}MPI core due to the modified symbol table. The core continues activating tool layers until the completion of all wrappers, after which it invokes the PMPI routine of the MPI library to complete the MPI call.

P^{N}MPI provides a low overhead method to create tool-ready executables. After linking their application with P^{N}MPI , users can change tool stacks merely by changing the configuration file. No additional relinking (or recompilation) is necessary.

Static Stacking

If the host OS does not support dynamic linking, P^{N}MPI must create the tool stacks before the application is linked with the MPI library. For this, we provide an offline stacking tool, which creates a static P^{N}MPI core, specialized to match the P^{N}MPI configuration file. Any application can then be linked with this specialized stack.

In order to avoid duplicate names caused by linking multiple PMPI tools as well as to redirect the execution after the completion of a wrapper back to the static core, we again use the binary rewrite tool, this time applied to the static version of the libraries. The patch utility adds a module specific prefix to all *MPI_* and *PMPI_* calls as well as all definitions of *MPI_* routines. This prefix is then used by the static stacking tool to identify the individual modules and tie them into the created stack at the appropriate levels.

Switch Modules still support switching between multiple stacks dynamically using with static stacking: only the stack contents must be defined at link time. Hence, the static version of P^{N}MPI provides the same functionality as its dynamic counterpart except that configuration changes require rerunning the static stacking tool and subsequent relinking.

3.3 Tool Initialization

Individual tool modules execute their initialization during the wrapper routine for *MPI_Init*, as it is typical for all PMPI tools. A P^{N}MPI -aware module must also provide an initialization routine that registers its services. The P^{N}MPI core probes for this routine in all modules and, if found, invokes it before calling the *MPI_Init* wrappers. Registration prior to those wrappers ensures that all queries return all available services.

4. BASE PERFORMANCE AND OVERHEAD

Tool infrastructures like P^{N}MPI must minimize application perturbation. In this section we extend the analytical overhead model for P^{N}MPI to include *Switch Modules*. We then calibrate the overhead through microbenchmarks and compare them for dynamic and static stacking. Our results demonstrate that both have negligible overhead although they are slightly lower with the slightly less flexible static stacking.

4.1 Modeling the Overhead

To model the overhead of P^{N}MPI, we must first consider the overhead of a single tool stack. Assuming N_{stack} tools in the stack, each wrapping all MPI function calls, and each tool has an execution time of $T_{tool}(i)$ for each function call, the overall tool stack execution time is:

$$T(stack) = \sum_{i=1}^{N_{stack}} (\beta + T_{tool}(i))$$

β represents the invocation overhead of P^{N}MPI for a single tool module. Since most tools wrap a small subset of MPI calls, the above formula is a strict upper bound on tool stack execution time.

A complete tool chain can consist of multiple stacks coupled by *Switch Modules*. Assuming the simplest case of three stacks A, B, C with N_A, N_B, and N_C tools respectively, coupled by a single *Switch Module* as shown in Figure 1 (right), the maximal execution time of the complete tool chain can be described as:

$$T_{chain} = T(stack_A) + T_{switch} + max(T(stack_B), T(stack_C))$$

Following the longest execution time decision path would extend this to multiple *Switch Module* tool chains.

For our purpose of measuring P^{N}MPI overhead, this simplified model ignores individual tool overheads:

$$T_{overhead} = \alpha_{stub} + \beta * (N_A + max(N_B, N_C)) + \gamma$$

α_{stub} describes the initial P^{N}MPI activation overhead of the stub library, while γ is the *Switch Module* overhead. Note that α_{stub} is zero with static stacking: it creates the P^{N}MPI core statically and, thus, does not require the stub library.

Closer examination of the stack switching code reveals that its execution sequence is essentially equivalent to continuing execution of the same stack. Thus, we can express γ as:

$$\gamma = \beta + T_{tool}(switch)$$

$T_{tool}(switch)$ describes the module-specific portion of the switch decision overhead, which we again ignore for our purpose of modeling P^{N}MPI overhead. Thus, the overall model is:

$$T_{overhead} = \alpha_{stub} + \beta * (N_A + max(N_B, N_C) + 1)$$

Based on this model and assuming similar communication behavior on all nodes (as it is typical for most SPMD MPI codes), we expect overhead linear in the number of tools loaded into P^{N}MPI plus a constant, which is independent of the number of MPI tasks. Hence, P^{N}MPI is generally neutral to the scaling behavior of applications. However, long tool stacks with significant execution times inside the tools themselves, as well as unbalanced stack sizes in scenarios with dynamic switch modules can lead to unbalanced system perturbation, which is known to affect scalability. The user must control this perturbation through careful configuration of the individual tools inside the MPI tool chain. We leave service modules that measure the perturbation and detect when it is large or unbalanced for future work.

4.2 Model Parameters

For the following experiments we use *Atlas*, an 1152 node, 44 TFlop/s peak capacity Linux cluster at Lawrence Livermore National Laboratory. Each node has four dual core, 2.4 GHz AMD Opteron processors and 16 GB main memory. An Infiniband network based on Mellanox hardware connects the nodes. Each node runs a full copy of Linux using a specialized version of RHEL 4 adapted for high performance computing. Thus, *Atlas* supports both dynamic and static stacking.

To validate the model and to determine the parameters α_{stub} and β we use a standard pingpong benchmark to measure MPI message latency. We vary the number of "empty" P^{N}MPI modules, i.e., modules that wrap all MPI calls, but do not perform any operation when activated. As a result, the execution time induced by them is equivalent to the overhead caused by their activation.

Our first experiment aims at calculating α_{stub} by comparing the latency of running the benchmark without linking to P^{N}MPI and with P^{N}MPI , but without loading any modules. In the latter, the stub library directly forwards any MPI call from the application to the MPI library without invoking the P^{N}MPI core. Our experiments show that this difference is below the system noise threshold for *Atlas* and hence effectively zero with dynamic stacking. Recall that α_{stub} is always zero with static stacking.

Figure 3 shows the results for latency measurements across a range of message sizes up to 4KB and with varying numbers of empty tool layers. As expected, each layer adds a roughly constant overhead across all message sizes by increasing the time spent during the invocation of the `MPI_Send` and `MPI_Recv` calls. The graphs also show that static stacking has slightly lower overhead than dynamic stacking, but behaves roughly similar. This observation is not surprising, since the code generated by the static stack creation tool is basically equivalent to the code included in the dynamic P^{N}MPI core, but contains fewer operations since it is created at compile time.

To further study this behavior, Figure 4 shows the latency for the minimal message size plotted against the number of tool layers as well as a linear curve fitted over the data. In both versions, the data matches the expected model, although system noise impacts static stacking more. The parameters used for fitting the curves result in a value for β of 61ns for the dynamic and 38ns for static stacking.

We arbitrarily chose the maximum number of fifty layers for these experiments to measure P^{N}MPI overhead. In reality we expect typical configurations of far fewer layers (under ten) resulting in negligible overall overhead, in particalur compared with the expected additional overhead of the actual tools. Alternatively, fifty is not an inherent limitation for P^{N}MPI: our stress tests have successfully used over 10,000 concurrent tool layers.

In addition to *Atlas* we have conducted similar experiments on several other machines including an x86-based cluster and a Blue Gene/L system. In all cases, the results were similar and matched the model presented above. As with *Atlas* the parameters for α_{stub} were negligible and the value for β were in the same order of magnitude.

4.3 Memory Overhead

Memory overhead is also a concern with 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 .

Dynamic Stacking Memory Overhead

We first model P^{N}MPI's memory usage under dynamic stacking. The model consists of three equations for for global variables, heap

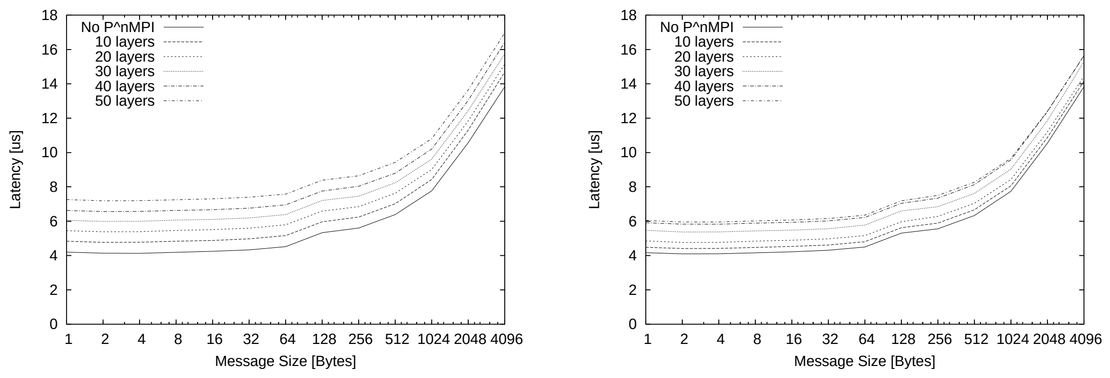

Figure 3: Latency with varying number of tool layers. Left: dynamic stacking; Right: static stacking.

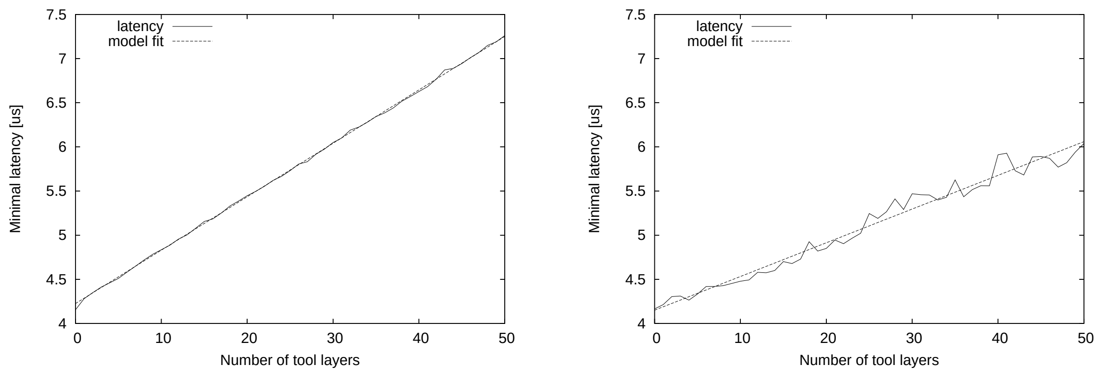

Figure 4: Minimal message size latency. Left: dynamic stacking; Right: static stacking.

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. In the following, we assume a 64-bit architecture, as is the case on *Atlas*. Note that the results for a 32-bit architecture would be reduced by almost a factor of two, since most stored values are pointers.

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 32 bytes for global configuration information, the following models global memory usage:

$$M_{global} = 32 + 8 * M + \lceil M/8 \rceil \ \ (Bytes).$$

Two data structures use the heap: a descriptor for each loaded module, which is currently 140 bytes, and the stacks for each MPI function intercepted 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 when at least one tool intercepts each MPI function:

$$M_{heap} \leq 140 * N + 8 * M * N \ \ (Bytes).$$

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

$$M_{stack} = (8 + 2 * Size_{stackframe} * (N + 1) \ \ (Bytes).$$

In our experiments on *Atlas*, we observe $Size_{stackframe} = 36$ and $M = 201$, so the total memory overhead is:

$$M_{overhead/dynamic} \leq 1896 + 1688 * N \ \ (Bytes).$$

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 per module.

Static Stacking Memory Overhead

Static stacking requires less global memory since it eliminates any state associated with dynamic stack management. Thus, static stacking does not consume any extra stack storage space and reduces the number of required additional stack frames to one. It also reduces global and heap memory usage to 20 Bytes of global information plus the 140 Bytes for module management discussed above. Thus, its complete memory consumption is:

$$M_{overhead/static} \leq 164 + 36 * N \ \ (Bytes).$$

Static stacking does not require tracking which MPI calls are intercepted since they are known when the offline stacking tool creates the specialized stack. Thus, the memory overhead of static stacking is independent of M and is, therefore, a good solution when most MPI calls are intercepted or for systems with limited memory per node, like Blue Gene/L.

5. P^{N}MPI CASE STUDIES

In this section we present three case studies that illustrate how to use P^{N}MPI's new capabilities to compose novel MPI tools: a message checksum computation tool, built directly from P^{N}MPI service modules; a communicator-specific MPI profiling tool, built on top of mpiP [12]; and an MPI environment virtualizer. These new tools are themselves useful. The checksum tool supports verification of MPI implementations. Communicator-specific profiling can significantly ease the effort required to optimize applications that use communicators to improve scalability, as is becoming increasingly common. Finally, the MPI virtualizer can provide an efficient, out-of-band communication mechanism to tools or support the dynamic combination of SPMD applications into one MPMD job.

For all following experiments we use dynamic stacking on *Atlas*. As shown in the previous section, performance with static stacking will generally be slightly better. We use only four cores in all experiments to minimize system noise. Finally, we present the minimum execution time from at least five runs per experiment.

5.1 Computing Checksums using a Stack of Tools

A message checksum tool can verify the correct transmission of messages by the MPI layer. To implement one, we must intercept all send events within an MPI application, compute checksums over the complete message data, and piggyback each checksum with the original message. Then, we must intercept the matching receive events, compute checksums over the complete received data and compare them to the piggybacked checksums.

Support Modules

We must track two central MPI data structures in order to compute message checksums. First, we need the type signatures of arbitrary MPI datatypes in order to traverse the message data. Second, we must associate request handles with their asynchronous operations in order to recreate all communication parameters at the completion of the message transfer during wait or test operations.

Many MPI tools must track these data structures. In general, each tool reimplements the services that provide this functionality due to the lack of modularity support in the PMPI interface. With P^{N}MPI , we implement each service in separate modules that is easily reused.

To traverse MPI datatypes, we implement a P^{N}MPI aware module that captures all MPI datatype creation operations and maintains a complete copy of this information. On request, the datatype module then provides a service for other MPI tools to create a handle for a given MPI message buffer as well as an iterator derived from a previously acquired handle to access all parts of the message buffer. Our checksum tool uses this mechanism to read the complete message contents and to compute the actual checksum.

The `request` module, the second service module that our checksum tool uses, allows tools to identify the corresponding communication initiation operation and its parameters during all MPI completion operations such as *MPI_Wait* or *MPI_Test*. The module intercepts all MPI operations that create MPI request objects, replaces these objects with its own copies and then uses these copies to store the context of their creation. This context includes the operation that initiated the request, the communication channel and the message parameters. Thus, MPI tools can retrieve any required context information during the corresponding completion.

However, we use an alternative mechanism to report the context to the tool since the MPI standard requires the destruction of request objects when the communication is completed. We use an additional module (`status`) to extend the MPI status object for this service. During its initialization, the `request` module uses this service to request additional storage space for the context information and then stores the context into a status object before returning from the respective test or wait operation.

```
#tool stack to create checksums

#extend status object
module status

#register datatypes
module datatype

#track communication and compute checksums
comm-checksum

#track requests
module requests
```

Figure 5: Configuration file to combine the necessary services to compute checksums.

Module to Intercept Communication

In addition to the service modules described above, we must intercept all MPI communication events. Thus, we wrap all MPI communication calls and each of these wrapper invokes the checksum routines. The need to wrap all communication events, however, is also common among many tools. Thus, we implement a generalized module that provides callbacks for each abstracted communication event independent of the actual MPI operation. This module abstracts all point-to-point (asynchronous and synchronous) and collective operations into a small set of routines, which the target tool can specialize for its particular purpose.

Our message checksum tool computes the checksums in the callbacks for send and receive completion events. It also transmits the checksum as a separate message in the send event callback and receives it in the receiver's callback for comparison. By using the same communicator and tag for this additional message, we are able to associate the checksum message uniquely with the original message, which is equivalent to piggybacking the information. This piggyback service is another common tool requirement. We are currently generalizing the mechanism to allow arbitrary payloads in a generic piggyback service module.

Using the services provided by the `request` and `datatype` modules and specializing the generic communication tracking module as describe above resulted in a highly modular, fully structured, and low complexity checksum tool implementation. In fact, its tool-specific module only requires 86 lines of code.

Configuration and Setup

Figure 5 shows the checksum tool configuration file. We first load the `status` module that extends the status objects. We load the `request` module last to make the extended request objects available in the entire tool stack. The remainder of the file specifies the loading of the `datatype` module and the tool-specific module (`comm-checksum`).

Results

We test our checksum tool with SMG2000 from the ASCI Purple benchmark suite, a Semicoarsening Multigrid Solver based on the hypre library [4]. It uses asynchronous send and receives in combination with *MPI_Waitall* for all communication. Also, it makes extensive use of custom MPI datatypes. Thus, it stresses the tool's performance.

We use input sizes of 100^3 on 4, 16, and 64 nodes with 4 cores each and compare the measured performance of the benchmark without P^NMPI, with each tool layer separately, and with the complete checksum tool. In addition, we use additional modules for communication tracking only (`comm`) and for piggybacking without checksum computation (`piggyback`) to further understand the overhead caused by the complete toolstack. Table 1 shows the results of these experiments.

The individual support modules alone incur only minimal overhead. Even the communication tracking module, which uses the other support modules to implement its functionality, exhibits a maximum overhead of 1.3%. Thus, both the P^NMPI infrastructure and its service modules only slightly impact performance. Also, their overhead is roughly constant across node counts, consistent with our analytical overhead model.

The majority of the overhead is caused by the actual tool. Piggybacking the additional data leads to an overhead between 2.8% and 5.5% and the complete checksum tool incurs a total overhead between 17.8% and 22.5%. We expect this higher overhead since it has to traverse each message at the sender and the receiver node to compute the checksums and piggybacking the checksums further results in twice the amount of messages during the application's execution. While such overhead is typically too high for performance analysis, it is often justified for debugging and correctness tools as it is the case for the checksum computation discussed here.

5.2 Enabling Multiple Scopes for Tools

Tools built using the PMPI interface are by default of global scope and hence get invoked on every MPI call. It is not easy to restrict them to parts of the application without explicitly adding this capability. A *Switch Module* easily supports this functionality without modifications to the tool. The *Switch Module* simply distinguishes the different application scopes and then activates a separate stack with an independent copy of the tool for each scope.

Example Scenario

To demonstrate this functionality, we use an example from the area of dense matrix computations. These applications often create separate communicators for row and column communication, as shown in Figure 6. Using a traditional profiler, like mpiP [12], would aggregate the communication behavior across all communicators and thereby hide the differences in row and column communication. Those differences can, however, provide necessary insight in order to optimize the application. Thus, the user could benefit from the option to profile each communicator set separately.

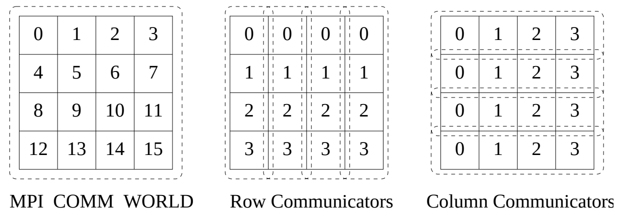

Figure 6: Using row and column communicators for matrix computations on a 4x4 matrix.

```
#define default stack
module commsize-switch
argument sizes 8 4
argument stacks row column
module mpiP

#define tool stack for rows
stack row
module mpiP1

#define tool stack for columns
stack column
module mpiP2
```

Figure 7: Configuration file to apply multiple copies of mpiP to individual application scopes.

Switch Module Design and Configuration

To achieve this functionality without having to hardcode it into a profiling tool, we use a P^NMPI switch module. It intercepts each communication call and then forwards the call into separate tool stacks depending on the size of the communicator[1] used for the particular MPI call. In each tool stack, we then run an unmodified, separate copy of mpiP, which collects a profile only for MPI calls forwarded to that particular stack.

Figure 7 shows our configuration file. We load the switch module with a `sizes` argument that applies different stacks to communicators of sizes eight and four (assuming a 32 task job mapped to a 8x4 processor grid). The former size is associated with the `row` stack, while the latter is associated with the `column` stack.

The rest of the configuration file then loads mpiP on the default stack as well as the `row` and `column` stacks. The different names of the mpiP instantiations represent copies of the mpiP library. These copies guarantee that the dynamic linker treats them as distinct libraries and allocates a separate global state for each.

Experiments and Results

Our experiments use QBox [5], a well-known and widely used First Principle Molecular Dynamics (FPMD) code developed at Lawrence Livermore National Laboratory. Its basic data structure is a large dense matrix and it uses ScaLAPACK/BLACS as well as BLACS directly to manipulate it. During the optimization of this code, the developers distinguished row and column communicators manually since the appropriate tool support did not exist. Further, the use of external libraries hides the use of row and column communicators and made profiling even more difficult.

Table 2 shows the profiling results for the top five MPI routine counts observed during an execution of QBox (using a working set size of 54 molecules on 32 tasks). The second column shows the results of a global profiling run, while the last three columns represent the results from our multiplexed tool stack (the third column is the sum of the last three, computed post-profiling). The results clearly show the different behavior for the three types of communicators used in the code: all-reduces and all-to-all calls are mainly restricted to `MPI_COMM_WOLRD`, while rows and columns dominate for point-to-point communication. Further, the code uses a

[1]We use the communicator size instead of its handle since many libraries, like BLACS, recreate the communicators multiple times leading to different handles in a single execution while the size remains constant.

Active	16 tasks / 4 nodes		64 tasks / 16 nodes		256 tasks / 64 nodes	
Modules	Exec. Time	Overhead	Exec. Time	Overhead	Exec. Time	Overhead
No P^{N}MPI	29.18	—	31.35	—	34.84	—
Status	29.28	0.3%	31.45	0.3%	34.98	0.4%
Requests	29.37	0.6%	31.59	0.8%	35.18	1.0%
Datatype	29.25	0.2%	31.42	0.2%	34.98	0.4%
Comm	29.47	1.0%	31.75	1.3%	35.30	1.3%
Piggyback	29.85	2.3%	32.48	3.6%	36.77	5.5%
Checksum	34.39	17.8%	37.11	18.4%	42.67	22.5%

Table 1: Execution times in seconds and overhead compared to the uninstrumented baseline for SMG2000.

Count	Global	Sum	COMM_WOLRD	Row	Column
Send	317365	317245	31014	202972	83259
Allreduce	319028	319028	269876	49152	0
All2allv	471488	471488	471488	0	0
Recv	379355	379265	93034	202972	83259
Bcast	401312	401042	11168	331698	58176

Table 2: Top five most called MPI routines during the execution of QBox. The minor difference in aggregate number of calls across both executions is caused by nondeterministic differences during the computation.

	Exec. Time	Overhead
Native	446.62	0.0%
mpiP	499.40	11.8%
P^{N}MPI & mpiP	499.88	11.9%
Multplexed	500.03	12.0%

Table 3: Execution times in seconds and overhead compared to the native baseline for QBox (using a working set size of 54 molecules) running with different versions of profiling (8 nodes, 32 tasks).

high number of broadcasts along rows. The standard profiling run does not reveal these differences and modifying mpiP to gather this data directly would have required significant changes.

Table 3 shows the execution times and overhead numbers for running the above experiments. We compare the native execution without tool support to using mpiP as a standard PMPI tool, using mpiP as a transparent P^{N}MPI module, and running our complete multiplexed tool stack. The results show that mpiP adds about an 11% overhead to the execution, while the differences between the various types of tool stacks is minimal.

5.3 MPI Virtualization

Many application scenarios require the cooperative execution of multiple MPI applications. A typical example for this are coupled simulations like in the *Cooperative Programming Model* [9] or in the Community Climate System Model (CCSM) [10]. Typically, these scenarios either use a framework specific MPI virtualization (as with CCSM) or rely on more loosely coupled mechanisms such as RMI [7]. Our virtualization module implemented in P^{N}MPI provides a generalized MPI virtualization that separates the execution environment of two or more independent, unmodified MPI applications.

Virtualization Module

To achieve virtualization, the module must first assign each rank to one application and then create separate communicators for the ranks assigned to each application. The module then uses these newly created communicators to replace *MPI_COMM_WORLD* in the virtualized applications. We wrap all communication calls that take a communicator as argument for this purpose. This automatically supports derived communicators since operations that originally derive new communicators from *MPI_COMM_WORLD* automatically use the restricted application-specific communicator instead.

After loading the virtualization module, applications no longer see the complete MPI partition, but are rather restricted to their own subcommunicators. Other additional modules loaded after the virtualization module, however, do have a global view of the complete MPI job and can be used to implement cross application communication services. This provides programmers of cooperative applications with a mechanism to exchange data between codes.

For the first step, the detection of the application ranks, our virtualization module provides two separate mechanisms: either ranks are assigned statically using a round robin or a block allocation scheme or ranks are dynamically detected at runtime by grouping binaries with the same filename (as provided by $argv[0]$). The former method can execute multiple, virtualized copies of the same application (e.g., for parameter studies), while the latter can execute two independent codes.

Configuration and Setup

Figure 8 shows several possible configuration files to enable the MPI virtualization. In all cases we only load the MPI virtualization module and parameterize it to assign the appropriate tasks to each application. The first two options show how to statically configure a 64 task job for two 32 task applications using round robin or block distributions respectively and the third option shows the configuration for dynamic task detection.

In the following we use the latter option and apply it to two separate applications: SMG2000, introduced above, and Sweep3D [1], a neutron transport code developed in FORTRAN 77.

Results

To evaluate the virtualization module, we compare the basic runtime of SMG2000 and Sweep3D with the execution time within the virtualization layer running both as single applications and in

SMG2000 Test Case	$N = 40^3$ time	$N = 40^3$ overh.	$N = 60^3$ time	$N = 60^3$ overh.	$N = 80^3$ time	$N = 80^3$ overh.	$N = 100^3$ time	$N = 100^3$ overh.	$N = 120^3$ time	$N = 120^3$ overh.
Native	2.14	—	6.03	—	13.96	—	30.71	—	52.15	—
Virtualized/Alone	2.20	2.6%	6.11	1.4%	14.08	0.9%	30.87	0.5%	52.39	0.5%
Virtualize/Combined	2.21	3.0%	6.11	1.3%	14.07	0.8%	30.88	0.5%	52.36	0.4%

Table 4: Execution times in seconds and overhead compared to the native baseline for SMG2000 with various working set sizes and running with and without virtualization (16 nodes, 32 tasks).

Sweep3D Test Case	$N = 100^3$ time	$N = 100^3$ overh.	$N = 150^3$ time	$N = 150^3$ overh.	$N = 200^3$ time	$N = 200^3$ overh.	$N = 250^3$ time	$N = 250^3$ overh.	$N = 300^3$ time	$N = 300^3$ overh.
Native	3.00	—	7.80	—	17.29	—	32.75	—	49.65	—
Virtualized/Alone	3.01	0.1%	7.81	0.0%	17.32	0.1%	32.75	0.0%	49.69	0.1%
Virtualize/Combined	3.01	0.1%	7.81	0.1%	17.32	0.2%	32.76	0.0%	49.61	-0.1%

Table 5: Execution times in seconds and overhead compared to the native baseline for Sweep3D with various working set sizes and running with and without virtualization (4 nodes, 32 tasks per application) — the one negative overhead value is most likely caused by system noise.

```
#virtualizing MPI jobs (round robin)
module virtual
argument jobs 2
argument tasks 32 32 round


#virtualizing MPI jobs (blocks)
module virtual
argument jobs 2
argument tasks 32 32 block


#virtualizing MPI jobs (by name)
module virtual
argument jobs name
```

Figure 8: Configuration files to virtualize an MPI job: 64 tasks split into two tasks using round robin (top) and block (middle) task distributions; based on the binary name (bottom).

combination. In the latter case we compare the performance of the applications without the final barrier during the termination of the virtualization module to disregard any differences in the overall runtimes of the two applications.

Tables 4, 5 show the execution times of SMG2000 and Sweep3D respectively. Overall, the overhead numbers for Sweep3D are significantly lower than for SMG2000. This is caused by the lower communication volume and frequency in Sweep3D compared to the communication intensive algorithm in SMG2000. However, the results always show that the virtualization overhead is very small and decreases with increased working set sizes, since setup and management overheads are amortized over a longer runtime. Further, no noticeable differences exist between the overhead combining the two applications and that of running either on top of the virtualization layer alone. Thus, virtualization works well with respect to functionality and performance.

	.C fi les	.H fi les	.W (wrapper generator) fi les
request	673	80	—
status	283	43	—
datatype	1165	171	—
comm-track	758	29	—
checksum	86	—	—
virtualization	135	—	10
multiplex switch	111	—	11
mpiP	14379	1427	—

Table 6: Lines of code in each module of our case studies, split into implementation files, header files, and MPI code wrapper generator descriptions, which automatically create wrapper code for subsets of MPI calls.

5.4 Discussion

These three case studies show that P^{N}MPI can be used to compose new MPI tool chains quickly and, in many cases, transparently for a variety of application scenarios. P^{N}MPI can even specialize or enhance existing tools, even if source code is unavailable, to provide application-specific tools.

Application-specific tools built with P^{N}MPI are not complex. To support this claim, we examine the number of lines required to implement each of the modules used during the case studies in Table 6. While only an approximate complexity metric, the results show that the complexity is in the actual tools, such as mpiP, and the support modules, which can be reused between tools. The experiment-specific modules, i.e., the extension of communication tracking to compute checksums, the switch module for matrix computation and the virtualization module, require only about one hundred lines of code. Further, those modules could also serve as templates for other application-specific tools. For example, we could easily implement a datatype-specific profiler based on the communicator-specific one. Thus, P^{N}MPI enables tool and application developers to prototype tools quickly by encapsulating MPI tool complexity in a few reusable support components.

6. RELATED WORK

We discuss here a small sampling of the many tools that use the PMPI interface to cover a wide range of application scenarios. mpiP [12] gathers profiling data about an application's communication usage. The TAU/ParaProf toolset [2] and Vampir NG [3] include MPI tracing tools for detailed performance analysis. UMPIRE [13] and Marmot [6] collect data for MPI correctness checking. Jitterbug [14] can be used to study and debug race conditions by randomizing nondeterministic communication paths.

These existing tools, however, only work in isolation. To our knowledge P^NMPI is the only infrastructure that enables the concurrent execution of multiple PMPI tools. Further, P^NMPI supports building on existing tools to create new application-specific tools. In particular, P^NMPI's ability to multiplex existing tools allows users to apply these tools selectively to dynamic parts of applications, significantly widening their scope without the need for modifications.

The general problem of tool interoperability has not been widely studied. To our knowledge, OMIS, the Online Monitoring Interface Specification [8], is the most comprehensive work on this topic. It defines services that allow arbitrary online tools to register and detect conflicts. Wismüller [15] discusses the various types of conflicts and proposes the use of couplers to resolve dependencies and enable clean tool-to-tool communication. In P^NMPI we face similar issues, but due to our restriction to PMPI message passing tools we can avoid many of these general problems. However, P^NMPI users must consider tool dependencies, in most cases by enforcing the correct order of the tools within the individual tool stacks.

7. CONCLUSIONS

P^NMPI is an advanced tool infrastructure that supports running multiple PMPI tools concurrently, without modification. In this paper, we have presented extensions to this infrastructure that significantly increase its power. Our extensions include a registration mechanism that supports the creation and use of tool service modules, which support the reuse of code across tools. We have also added *Switch Modules*, which enable users to apply unmodified existing tools selectively. Finally, our MPI virtualization module provides virtualized MPI environments to construct MPMD-style applications. Further, we modified the P^NMPI core to support static stacking as well as dynamic stacking, making P^NMPI available on platforms, like BG/L, that lack dynamic linking support.

We extended the existing P^NMPI performance model to account for our improvements. We calibrated the model on a large scale Linux cluster and showed that dynamic and static stacking have negligible overheads. More importantly, we presented three case studies with extensive performance results. These case studies, which implement useful MPI tools, demonstrate the power of our P^NMPI extensions. Overall, P^NMPI provides a flexible, highly productive environment that supports the design and implementation of application-specific MPI tools.

8. REFERENCES

[1] Accelerated Strategic Computing Initiative. The ASCI sweep3d benchmark code. http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/, December 1995.

[2] 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, August 2003.

[3] H. Brunst, D. Kranzlmüller, and W. Nagel. Tools for Scalable Parallel Program Analysis - Vampir NG and DeWiz. *The International Series in Engineering and Computer Science, Distributed and Parallel Systems*, 777:92–102, 2005.

[4] R.D. Falgout and U.M. 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, April 2002.

[5] F. Gygi, E.W. Draeger, M. Schulz, B.R. de Supinski, J.A. Gunnels, V. Austel, J.C. Sexton, F. Franchetti, S. Kral, J. Lorenz, and C.W. Überhuber. Large-Scale Electronic Structure Calculations of High-Z Metals on the BlueGene/L Platform. In *Proceedings of IEEE/ACM Supercomputing '06*, November 2006.

[6] 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*, September 2005.

[7] G. Kumfert, J. Leek, and T. Epperly. Babel remote method invocation. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium*, March 2007.

[8] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. *OMIS — On-line Monitoring Interface Specifi cation (Version 2.0)*, volume 9 of *LRR-TUM Research Report Series*. Shaker Verlag, Aachen, Germany, 1997. ISBN 3-8265-3035-7.

[9] J. May, D. Jefferson, N. Barton, R. Becker, J. Knap, G. Kumfert, J. Leek, and J. Tannahill. Introducing Cooperative Parallelism. Presented at the CCA Forum, presentation available at http://www.cca-forum.org/download/mtg/2007-01/may-coop-cca.ppt, January 2007.

[10] National Center for Atmospheric Research (NCAR). Community Climate System Model (CCSM). http://www.ccsm.ucar.edu/, 2006.

[11] M. Schulz and Bronis R. de Supinski. A Flexible and Dynamic Infrastructure for MPI Tool Interoperability. In *Proceedings of the 2006 International Conference on Parallel Processing*, August 2006.

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

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

[14] R. Vuduc, M. Schulz, D. Quinlan, B. de Supinski, and A. Sæbjørnsen. Improving Distributed Memory Applications Testing By Message Perturbation. In *Proceedings of Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, July 2006.

[15] R. Wismüller. *Interoperable Laufzeit-Werkzeuge für parallele und verteilte Systeme*. Inaugural dissertation (Habilitation), Fakultät für Informatik, Technische Universität München, München, Germany, August 2001.