Runtime Techniques for Automatic Process Virtualization

Evan Ramos
Charmworks Inc.
Urbana, IL, USA
evan@hpccharm.com

Sam White
Department of Computer Science
The University of Illinois at Urbana-Champaign
Urbana, IL, USA
white67@illinois.edu

Aditya Bhosale
Department of Computer Science
The University of Illinois at Urbana-Champaign
Urbana, IL, USA
adityapb@illinois.edu

Laxmikant V. Kale
Department of Computer Science
The University of Illinois at Urbana-Champaign
Urbana, IL, USA
kale@illinois.edu

ABSTRACT

Asynchronous many-task runtimes look promising for the next generation of high performance computing systems. But these runtimes are usually based on new programming models, requiring extensive programmer effort to port existing applications to them. An alternative approach is to reimagine the execution model of widely used programming APIs, such as MPI, in order to execute them more asynchronously. Virtualization is a powerful technique that can be used to execute a bulk synchronous parallel program in an asynchronous manner. Moreover, if the virtualized entities can be migrated between address spaces, the runtime can optimize execution with dynamic load balancing, fault tolerance, and other adaptive techniques.

Previous work on automating process virtualization has explored compiler approaches, source-to-source refactoring tools, and runtime methods. These approaches achieve virtualization with different tradeoffs in terms of portability (across different architectures, operating systems, compilers, and linkers), programmer effort required, and the ability to handle all different kinds of global state and programming languages. We implement support for three different related runtime methods, discuss shortcomings and their applicability to user-level virtualized process migration, and compare performance to existing approaches. Compared to existing approaches, one of our new methods achieves what we consider the best overall functionality in terms of portability, automation, support for migration, and runtime performance.

ACM Reference Format:
Evan Ramos, Sam White, Aditya Bhosale, and Laxmikant V. Kale. 2022. Runtime Techniques for Automatic Process Virtualization. In *51th International Conference on Parallel Processing Workshop (ICPP Workshops '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3547276.3548522

ICPP Workshops '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9445-1/22/08…$15.00
https://doi.org/10.1145/3547276.3548522

1 INTRODUCTION

As high performance computing systems continue to evolve into exascale, application developers are being asked difficult questions. Can their code scale up to exascale? Can it make use of heterogeneous compute nodes? What about different memory technologies or network interconnects? Can it tolerate network latency given the increased relative costs of data movement? Can it use multi-scale or other dynamic methods to increase simulation resolution only where needed, in areas of interest? At the same time, different questions are being posed by another class of emergent computing systems– the cloud. Is your application readily deployable in the cloud? Can it optimize performance for the different cluster configurations there? What happens if the price of compute resources changes during a run– can the job be stopped and restarted from that point later on? What if a hardware node fails during the run?

Concurrent with these hardware advances, asynchronous many-task runtime systems are proving their ability to manage resources dynamically in response to changing application and hardware behaviors. Task-based programming systems such as Legion [3], HPX [14], Charm++ [1], and Chapel [6] are able to monitor performance during execution and introspectively adapt execution on the fly. They do so in part by taking advantage of different programming models that have been designed from the ground up with asynchrony, heterogeneity, and fault resilience in mind. Their unique programming models also dictate, however, that porting any existing code to them requires significant programmer effort. Serial portions of code that constitute the lowest level tasks can usually be left as is, but the parallel control flow must be rewritten. For legacy codes, this is often a non-starter due to code complexity and the developer effort required. Consequently, effort has been put into making MPI interoperation work with these tasking models, though that still requires part of the application to be rewritten, and then execution has to be handed off between MPI and the tasking runtime's scheduler which requires synchronization [13].

An alternative approach is to reimagine the execution model of existing parallel programming models or libraries, such that the existing code can be run on the tasking runtime system. MPC [18] and Adaptive MPI [12] are examples of such an approach applied to the MPI programming interface [10]. MPI is widely used in the high performance computing domain and as of now lacks high-level, easy to use support for some of the key features that tasking

runtimes offer such as efficient integration with shared memory programming models, dynamic load balancing, and fault resilience. In order to fulfill this promise, these runtimes rely on the technique of *process virtualization*.

Process virtualization, as we define it, refers to the abstraction of operating system processes such that code meant to run as a process can instead be run on a thread. For example, a code that uses a mutable global variable (such as in figure 2) can not be virtualized as is, since the global variable itself will be shared among multiple threads. We refer to the process of converting a code, either manually or automatically, to a virtualizable code as *privatization*. Process virtualization allows abstracting the notion of an MPI rank, which typically in a library-based MPI implementation is equivalent to a process, instead creating and running multiple MPI ranks as user-level threads within each process. The runtime system then manages the scheduling and communication between ranks, even supporting dynamic migration of ranks across address spaces at runtime. Process virtualization is, consequently, key to enabling adaptive runtime features to work on a legacy MPI application, and fully automatic privatization support is a common goal of runtimes like AMPI or MPC.

Previous work has attempted to automate process virtualization, with differing degrees of success. In this work, we summarize the state of the art privatization methods, discuss advantages and limitations of each, and then implement support for three novel runtime privatization methods in AMPI, the last of which achieves a new degree of applicability to legacy codes with greater portability across compilers and linkers, high performance in multiple aspects, and support for advanced features such as dynamic rank migration. This work is important in making the advanced runtime features of AMPI and others like it available to legacy applications that need them.

2 BACKGROUND

2.1 Adaptive MPI

Adaptive MPI is an implementation of the MPI standard developed on top of Charm++ and its dynamic runtime system. Ranks are virtualized as migratable user-level threads rather than processes, so that users can run with more ranks than cores, i.e. overdecomposition, as shown in Figure 1. Overdecomposition enable message-driven cooperative scheduling of ranks on each core. This works by having a rank suspend its user-level thread when encountering a blocking communication call that it cannot immediately fulfill: instead of busy-waiting on the network, the scheduler context switches to another rank if one is ready to execute. User-level thread (ULT) context switches are fast (~100 nanoseconds), significantly faster than network communication latency.

Overdecomposition, combined with migratability of ranks, enables dynamic load balancing. The runtime can monitor performance metrics such as execution time per rank, idle time per PE, the communication graph, and more in order to make rebalancing decisions. The main benefit of this kind of rebalancing based on dynamic rank migration is that the rebalancing logic is separate from the application logic: rank's do not have to be aware of their placement at any given time since the Charm++ runtime performs efficient distributed location management of ranks. There is also no user serialization code needed to enable migration, since AMPI's Isomalloc memory allocator (inspired by [2]) completely automates migrating the user-level thread stack and heap.

AMPI relies on rank or process virtualization for latency hiding, dynamic load balancing, and dynamic job shrink/expand. Privatization, then, is key to making AMPI usable by legacy codes. If a user is writing an AMPI program from scratch, they can simply write an MPI program that does not use mutable global state and it will be virtualizable without any need for de facto privatization. However, the majority of users either already have MPI applications or want to use existing libraries in their application. Thus, automating code privatization as much possible is highly desirable.

2.2 Privatization Issues

To make privatization more concrete, Figure 2 provides a small code sample of an unsafe MPI program and a possible output of executing it in a virtualized manner (with multiple ranks in the same OS process). In the example output here, note that the zeroth rank sets the global variable *my_rank*'s value to its rank number 0, then blocks in the *MPI_Barrier()* call. Next, rank 1 will be scheduled and set *my_rank*'s value to its rank number 1 before suspending in the barrier. When both ranks are awakened from the barrier's completion in some ordering, they will both print the value of the last rank's number instead of their own, as shown in Figure 3. Any MPI user would expect this program to output "rank: 0" and "rank: 1" in some order, and this discrepancy would lead to correctness issues in a more complex code. This is because the variable is global, and global variables are defined in a per-process manner. Static variables are similarly defined per-process and suffer from the same issues in terms of virtualization.

Examples of unsafe variables are, in C/C++, non-const global and static variables. In Fortran, implicit or explicit save variables are static, and non-parameter module variables and common blocks are examples of global variables. We note that global variables whose value is written only once to the same value across all ranks are actually safe, since their value can be shared across all ranks. This is true of *num_ranks* in Figure 2. Thread unsafe virtualization issues arise when ranks are writing different values to the same variable, and that is why privatization is needed. And since legacy MPI applications and libraries can contain hundreds or thousands of such mutable global/static variables spread throughout the code in a pervasive manner, automatic privatization is essential.

2.3 Existing Privatization Techniques

Here we survey the state of the art privatization techniques. They vary in degrees of automation; applicability to different programming languages and kinds of variables; portability across compilers, linkers, operating systems, and architectures; performance in terms of runtime and memory overhead; and in support for migratability of data.

2.3.1 ***Manual code refactoring.*** is what we call the process of rewriting a code so that it does not use mutable global state. This usually requires encapsulating all global/static variables in an application into one or more structures which can then be allocated on the stack or heap and pointers to it passed around to all functions that reference the state. It also involves avoiding the use of

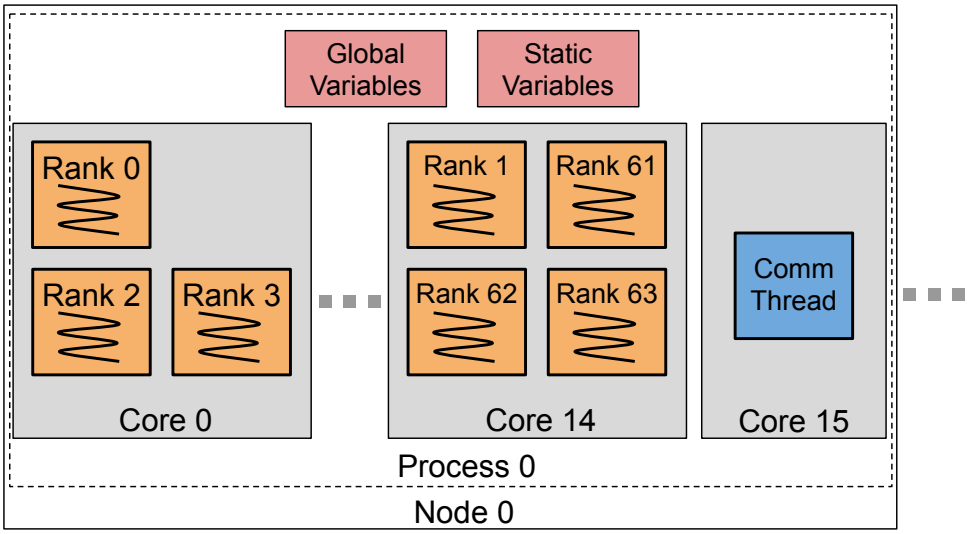

Figure 1: AMPI applications typically run with more virtual ranks than cores or PEs. Each virtual rank has its own user-level thread stack and heap memory. Notice that all ranks here share the same set of global and static variables by default. Privatization is necessary to provide each rank with their own separate copy of these variables. Also note that dynamic load balancing can alter the mapping of ranks to cores during execution. Here rank 1 has migrated from core 0 to core 15 on node 0. In practice, there are often multiple virtual ranks per core, and one OS process per socket or node. Ranks can even migrate across nodes.

```c
#include <mpi.h>
#include <stdio.h>

int my_rank;
int num_ranks;

int main(int argc, char** argv)
{
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_ranks);

  MPI_Barrier(MPI_COMM_WORLD);
  printf("rank: %d\n", rank_global);

  MPI_Finalize();
  return 0;
}
```

Figure 2: MPI hello world program in C with global variables.

```
$ ./hello_world +vp 2
rank: 1
rank: 1
```

Figure 3: Possible output from executing the MPI hello world program (above) with 2 Virtual Processors (VPs) in 1 OS process.

thread unsafe library calls, such as C/C++ *strtok* and *getopt*. If a code only contains a few such variables that are rarely referenced, this manual code refactoring process can be doable, since the changes themselves are simple and mostly mechanical to make. However, oftentimes in legacy codes the effort required is significant due to the use of hundreds or thousands of global/static variables being used throughout the code.

2.3.2 ***Source-to-source code refactoring tools.*** can automate the tedious refactoring process described above. Photran [16] [17] was developed for Fortran codes as an Eclipse plug-in that worked on Abstract Syntax Trees of the code. It encapsulated all global/static variable references into a single Fortran derived type (equivalent to a C structure), and passed that structure to all functions that referenced the global/static variables. These methods have much promise, since altering the source code remains the most portable method of privatization, so long as the solution can work on various programming languages and ideally incorporate programmer input on grouping variables into multiple structures rather than a single large structure for the entire codebase. We also note that this method can be combined with other methods which are semi-automatic, such as TLSglobals (described below).

2.3.3 ***Swapglobals.*** relies on details of the ELF object format to automatically privatize global variables [23]. ELF maintains a Global Offset Table of all global variables, and the table can be swapped out when context switching ranks at runtime. This method does not require any changes to the source code and works with Fortran and C/C++ code. However, it does not handle static variables, since they are not stored in the global offset table. It only works on x86 architectures that fully support ELF, and it requires either a version of the ld linker 2.23 or older or a patched version of ld 2.24 or newer in order to avoid the linker optimizing out the GOT pointer reference at each global variable access. Additionally, it does not

work in AMPI's SMP mode (illustrated in figure 1), since there can only be 1 GOT actively in use per OS process at a given time, whereas SMP mode has multiple user-level schedulers running concurrently on each core within an OS process. The non-SMP mode restricts AMPI to having one process per core, rendering some of its optimizations for shared address space communication ineffective [22]. This, combined with its limited portability and applicability to static variables, led to Swapglobals being deprecated.

2.3.4 ***TLSglobals***. depends on the user tagging their mutable global/static variable declarations with the *thread_local* attribute [23]. The runtime then switches out the TLS segment pointer at each user-level thread context switch. This method works on C/C++ and Fortran (using the *__thread* attribute in C, *thread_local* in C++, and OpenMP's *threadprivate* directive in Fortran), and on static and global variables alike. So far, it only works on Linux and Mac operating systems and on x86 architectures, though it could work on others. It also requires GCC or a recent version of the Clang compiler (v10.0+), because it requires support for their *-mno-tls-direct-seg-refs* option which forces the compiler to access TLS variables through the segment pointer always. As such, it introduces an indirection to each privatized variable access, which can result in performance degradation. We have extended TLSglobals to work with shared object linking and to work on ARM and Power architectures as well.

2.3.5 ***Compiler automated TLS variable tagging***. has been developed for MPC with support in the Intel compiler and with patched version of GCC available [4]. The user specifies a compiler option (*-fmpc-privatize*) which tells the compiler to automatically treat all global/static variables as if they were declared as a thread-local variables. It removes the need for users to identify and tag all unsafe variable declarations, and otherwise performs at runtime like TLSglobals. The runtime performance in turns depends on TLS variable access being as fast as access to unprivatized variables, which is architecture specific. This method also requires access to all dependent libraries in source code format so they can be recompiled with the special compiler support. MPC also includes support for hierarchical local storage, with additional attributes defined for data that needs to be privatized to varying levels of the node, core, ULT, or task hierarchy in order to minimize memory overhead [21].

2.3.6 ***Process-in-Process (PIP)***. is a user-level library developed by Hori et al that can be used to create a shared address space between processes [11]. It has been used primarily to share memory at the user-level between multiple MPI processes resident on the same node for fast intranode communication. To the best of our knowledge it has not been used for MPI rank virtualization or overdecomposition in which multiple ranks are co-scheduled on each PE and can dynamically migrate between nodes, but we recognized its applicability to this execution model. It relies on compiling all code into a Position Independent Executable (PIE). Position independent executables define their contents (including global and static variables) as offsets from the instruction pointer, so that the executable can be loaded into an arbitrary location in virtual memory. This is the default on most modern operating systems for security reasons. PIP then relies on the glibc-specific, non-standard function *dlmopen* which can be called on the PIE binary with a unique namespace index to duplicate all code segments, including the global variables. This method has no compiler requirements (except for support for PIE, which is ubiquitous) and does not require any programmer effort, however, it does require glibc and a patched version of glibc at that in order to create more than 12 namespace indices (virtualized entities) per OS process. This is a seemingly arbitrary limit inside glibc's implementation and so PiP distributes patched versions of glibc along with its source code to get around this limitation.

2.4 Shortcomings of Existing Approaches

Existing approaches fall broadly under source code modification tools, compiler-based approaches, or runtime methods, with some combining elements of both. We can categorize and evaluate the existing privatization methods based on several criteria. One being the degree of automation or amount of application programmer effort required; another being portability across compilers, linkers, and operating systems; third being support for many scheduler threads per OS process, each with their own virtualized entities; and finally, support for runtime migratability of virtualized entities between address spaces.

Table 1 summarizes our review of existing privatization methods by rating each in terms of these criteria. Which criteria are important to a particular user will vary by their requirements.

3 AUTOMATIC RUNTIME PRIVATIZATION

Based on our desire to avoid portability restrictions such as requiring specific compilers while fully automating privatization, we found the runtime techniques at the heart of the Process-in-Process library appealing. This led to us developing support in Adaptive MPI for three new privatization methods. All three compile the application as a Position Independent Executable and duplicate the code segments for each virtual rank in a process. They differ primarily in how they duplicate the code, and this has significant consequences for portability, performance, and the ability to dynamically migrate ranks.

3.1 PIPglobals

We initially looked into integrating the Process-in-Process library into AMPI for the purpose of automatic global/static variable privatization. After discussion with its developers, we found that AMPI did not need to use the PIP library directly, and instead we have applied concepts from PIP and implemented the parts that we need from it for our purposes inside AMPI. Doing so also allowed us to target more architectures than PIP did at the time and to streamline the startup process by tailoring PIP's internals to AMPI's specific needs. We call this method PIPglobals.

This allowed us to construct a prototype that works for AMPI privatization. It works by compiling the user program as a Position Independent Executable and linking it against a special shim of function pointers. We can not simply compile the application as a PIE and then call *dlmopen*, because that would lead to the AMPI runtime system being privatized along with the application code. Instead, we need to privatize only the application while running multiple copies of its code on a single copy of the runtime per OS

Table 1: Summary of existing privatization methods and their features.

Method	Automation	Portability	SMP Mode Support	Migration Support
Manual refactoring	Poor	Good	Yes	Yes
Photran	Fortran-specific	Good	Yes	Yes
Swapglobals	No static vars	Linker-specific	No	Yes
TLSglobals	Mediocre	Compiler-specific	Yes	Yes
-fmpc-privatize	Good	Compiler/linker-specific	Yes	Not implemented, but possible
Process-in-Process	Good	Requires GNU libc extension	Limited w/o patched glibc	Unknown

process. To do so, we refactored AMPI's headers into a function pointer shim library that the application links against. At startup, a small loader utility then calls the glibc-specific function *dlmopen* on the user's PIE binary with a unique namespace index for each virtual rank. The loader uses *dlsym* to locate a special function linked with the user's binary, passes it a structure with pointers to the entire AMPI API in order to populate the PIE binary's function pointers. Then it locates and calls the entry point. This *dlmopen* and *dlsym* process repeats for each rank. As soon as execution jumps into the PIE binary, any global variables referenced within it appear privatized. This is because PIE binaries locate the global data segment immediately after the code segment so that PIE global variables are accessed relative to the instruction pointer, and because *dlmopen* creates a separate view of these segments in memory for each unique namespace index. This also means that there is no work to be done at user-level thread context switch time, and the cost of accessing global data should be the same as in the unprivatized code. We anticipated the startup overheads being insignificant for practical degrees of virtualization (typically in the range of tens of cores per OS process and roughly ten virtual ranks per core).

We did encounter two limitations. First, we can not support high degrees of virtualization without using the patched version of glibc provided with the PIP library. This particularly limits the utility of the method in SMP mode. Second, we have not been able to implement support for runtime rank migration, which means an AMPI program virtualized via PIPglobals can not perform dynamic load balancing, checkpoint/restart-based fault tolerance, etc. This is because we cannot intercept the *mmap* calls that happen from inside *ld-linux.so* in order to allocate them via Isomalloc, AMPI's migratable memory allocator. Also we are restricted to GNU/Linux systems due to the reliance on non-POSIX-standard *dlmopen*.

```
// In ampi_functions.h:
AMPI_FUNC (int, MPI_Send, const void *msg, int count,
     MPI_Datatype type, int dest, int tag, MPI_Comm comm)

// In mpi.h:
#ifdef AMPI_USE_FUNCPTR
 #define AMPI_FUNC(return_type, func_name, ...) \
 extern return_type (* func_name)(__VA_ARGS__);
#else
 #define AMPI_FUNC(return_type, func_name, ...) \
 extern return_type func_name(__VA_ARGS__);
#endif
#include "ampi_functions.h"

// In ampi_funcptr.h:
struct AMPI_FuncPtr_Transport {
 #define AMPI_FUNC(return_type, func_name, ...) \
 return_type (* func_name)(__VA_ARGS__);
 #include "ampi_functions.h"
};

// In ampi_funcptr_loader.C (linked with AMPI runtime):
void AMPI_FuncPtr_Pack (struct AMPI_FuncPtr_Transport *x)
      {
 #define AMPI_FUNC(return_type, func_name, ...) \
 x->func_name = func_name;
 #include "ampi_functions.h"
}

// In ampi_funcptr_shim.C (linked with MPI user program):
void AMPI_FuncPtr_Unpack (struct AMPI_FuncPtr_Transport *
     x) {
 #define AMPI_FUNC(return_type, func_name, ...) \
 func_name = x->func_name;
 #include "ampi_functions.h"
}
```

Figure 4: AMPI's headers had to be refactored as a function pointer shim library to avoid privatizing its runtime along with the user application code.

3.2 FSglobals

FSglobals takes the same idea of PIPglobals but instead of relocating the code in memory we copy it onto a shared file system. This has two main benefits, and one drawback. It makes this method portable beyond GNU/Linux systems and removes the limit of creating 12 virtual ranks per OS process. At the same time, it does require a shared file system and space on that file system for each virtualized rank's copy of the binary. This can cause FSglobals to be slow during startup as well, with all the I/O involved. In particular, we did not expect it to scale well to large runs with many virtual ranks. Overall, it works similarly to PIPglobals but instead of calling *dlmopen* with namespaces we create copies of the PIE binary on the file system and call the POSIX-standard *dlopen*.

We also note that shared objects are currently not supported by FSglobals due to the extra overhead of iterating through all dependencies and copying each one per virtual rank while avoiding system components, plus the complexity of ensuring each rank's program binary sees the proper set of objects.

FSglobals unfortunately suffers from the same issue as PIPglobals of not being able to intercept the code segment copies during initialization. This means that FSglobals does not support dynamic rank migration either.

3.3 PIEglobals

We developed a third related privatization method in order to support migration with privatization, which we call PIEglobals. We consider it the most fully automated method we have so far. As with

PIPglobals, this method builds the user's MPI program as a shared object in Position Independent Executable mode. After initialization of the AMPI runtime, execution is handed off to a loader utility that performs PIEglobals setup. It first opens the PIE shared object using the system's dynamic linking capabilities, and then calls the glibc extension *dl_iterate_phdr* before and after the *dlopen* call in order to determine the location of the PIE binary's code and data segments in memory. This is useful because PIE binaries access global variables relative to the instruction pointer, and they locate the data segment immediately after the code segment. Our PIEglobals loader makes a copy of the program's code and data segments for each AMPI rank in the job via the Isomalloc allocator, thereby privatizing their global state while also ensuring that memory can be migrated across address nodes. It then constructs a synthetic function pointer for each rank at its new locations and calls it.

No further effort is required of the user to achieve global variable privatization beyond building their program through AMPI's toolchain wrappers with the *-pieglobals* compiler option. Migration (for load balancing or fault tolerance) works because the code and data segments have already been allocated via Isomalloc. Any libraries and shared objects compiled as PIE will also be privatized. The technique is broadly portable to GNU/Linux systems, since all necessary glibc extensions have existed in stable releases of it since 2005. We have validated it on x86, ARM, and POWER architectures.

This method did present a series of development challenges. For one, shared objects maintain tables of function addresses (the Global Offset Table) used by the machine code for indirect lookups to other code, no matter where in memory the code is located, and they must be updated when PIEglobals moves it to a different location. Currently this is done by scanning memory inside the data segment boundaries identified by glibc for contents that look like pointers to the code's original location, which we intend to replace with a more robust method unaffected by false positives. Similarly, C++ codes can contain global variables that are initialized at startup using class constructor methods, which sometimes make heap allocations. With PIEglobals, this takes place at the time *dlopen* is called on the user's binary, before any interception and privatization can take place. These allocations must be logged, replicated for each privatized rank created, their contents copied. It is possible for any arbitrary data written by these constructors to contain pointers to other globals or heap allocated data, as well as function pointers (particularly in classes with virtual functions), which must also be updated. We also found that it is important to open the shared objects only once per OS process, rather than once per virtual rank, in SMP mode to avoid crashes in glibc due to interactions between *dlopen* and pthreads. Finally, we had to ensure that TLS variables inside applications and system libraries are privatized correctly with PIEglobals to each virtual rank by combining the method with TLSglobals.

Another challenge arose inside AMPI when using PIEglobals: anywhere that AMPI previously relied on a function pointer being the same across ranks would break now that each rank had its own unique copy of the code. AMPI implemented user-defined custom reduction operators by simply calling the same user function pointer on whichever core it may need to. With PIEglobals, we had to modify AMPI to subtract the base address from the user function address during *MPI_Op* creation, to store that offset in the op, and to then apply that offset to some rank's base address whenever applying the reduction operator. Since virtual ranks can migrate around the system arbitrarily, however, it is possible for a core to have no virtual ranks assigned to it at a given time. It is possible then that the runtime would be processing a user-defined reduction and a core that has no virtual ranks resident on it would like to combine reduction messages. While we could modify AMPI's reduction algorithms to be aware of empty cores and avoid processing reduction contributions on them, instead forwarding the messages, since it is rare in practice to have an empty core, we instead require that all cores have at least virtual rank assigned to them during reduction processing with PIEglobals enabled and otherwise throw a runtime error when the reduction can't be processed.

Debugging code privatized with PIEglobals can also be difficult because debuggers such as GDB and LLDB do not know what debug symbols correspond to the manually copied code segments, leaving backtraces mostly mysterious. For this reason we provide a faculty to assist in debugging with virtualization, called *pieglobalsfind*. This can be called at runtime from within a debugger to translate a privatized address back to its original location as allocated by the system's runtime linker, thereby associating it with any debug symbols included in the binary.

Performance-wise, we had three initial concerns when developing PIEglobals. First, we theorized that copies of the code segment might cause instruction cache misses due to redundant copies of code being used separately by each rank. Second, we were concerned that startup costs would be significant due to the need for copying all code segments and then scanning for function pointers in any heap allocated static objects and updating them to point to the privatized code segment. Third, we knew that for large codes the migration overhead would be increased since the code segments must be migrated along with all the rank's heap-allocated memory and its user-level thread stack. In turn, this could make load balancing more costly. We discuss these aspects further in the performance evaluation and future work sections.

4 RESULTS

We looked at the runtime performance of our three methods and compared them against other existing methods. We note that runtime performance is but one criteria– among portability, developer effort, suitability to dynamic migration, and maintainability of code– to consider when evaluating privatization methods for an application, but since our three new methods are all runtime-based, the overheads must be kept reasonably low in order for them to be effective.

We break down performance into multiple different aspects: startup or initialization time, context switch overhead, privatized variable access overhead, and migration overhead. Startup or initialization time is a one-time cost for each program execution, while privatized variable access and context switch times are expected to be paid many times per program execution. Migration is typically infrequent– done in reaction to dynamic load imbalance or hard faults– so its price is expected to be paid less often than, say, privatized variable access.

We used the Bridges-2 supercomputer at the Pittsburgh Supercomputing Center for all of our performance measurements.

Bridges-2 is comprised of three different node type partitions: regular memory, extreme memory, and GPU. We use the "regular memory" nodes, each node having 2 AMD EPYC 7742 CPUs with 64 cores and 256 GB of memory. We used GCC v10.2.0 and Charm++'s MPI networking layer (OpenMPI v4.0.5) on the Mellanox Infiniband network interconnect. We compare the performance of AMPI's existing method TLSglobals against our three new ones: PIPglobals, FSglobals, and PIEglobals. We were unable to get Swapglobals working on this system for comparison.

4.1 Startup overhead

We first measure the time spent in AMPI initialization. For the various privatization methods, we generally expected the runtime-based ones to incur higher overhead here. In particular, our three new methods duplicate the code segments of the application binary once per virtual rank in each OS process at startup. Compared to TLSglobals, which only has to copy the TLS segment once per virtual rank per process, we expected our new methods to perform worse. That being said, since the initialization or startup overhead is only paid once per job, some overhead can be tolerated as long as it does not scale up with the node count.

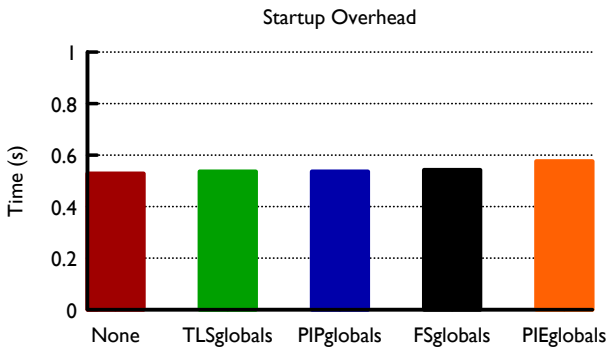

Figure 5: Startup or initialization overhead for different privatization methods with 8x virtualization (lower is better).

Figure 5 shows the startup time performance for each privatization method for 8 virtual ranks per process. The worst of our new methods performs 9% worse than the baseline without privatization. We note that with the exception of FSglobals, which relies on a shared file system, the cost is constant per-process and does not increase with node counts.

4.2 Context switch overhead

We next measured the time spent per user-level thread context switch. This is important because AMPI and runtimes like it use overdecomposition to hide latency via message-driven scheduling–increases in scheduling overhead can harm strong scaling performance and limit the degree of profitable overdecomposition. Higher degrees of overdecomposition are desirable for performing efficient dynamic load balancing.

We wrote a microbenchmark that context switches between two different user-level threads with each different privatization technique. Figure 6 shows the results averaged over 100,000 context

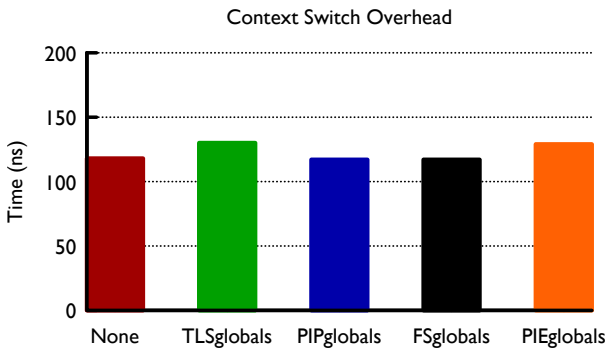

Figure 6: User-level thread context switch time in nanoseconds for each privatization method (lower is better).

switches. They demonstrate that TLSglobals and PIEglobals perform worst, but we note that all methods measured are within 12 nanoseconds of the baseline (without any privatization technique). This overhead is very small and does not increase based on the number of global variables or code size for any of the methods. Note that the time includes scheduling costs as well, since each time a ULT yields, control returns to the scheduler which then context switches to the next ULT that is ready to execute. TLSglobals requires updating the TLS segment pointer at each context switch, while the rest of the privatization techniques do not rely on any additional work at context switch time, since their global variables are defined relative to the instruction pointer. We do note that PIEglobals implies use of TLSglobals where supported, so it includes the overhead of updating the TLS segment pointer. Hence, it is not surprising that TLSglobals and PIEglobals perform worst here, but we deem their minor overhead acceptably low.

4.3 Privatized variable access overhead

Another important characteristic of a privatization method is that the time spent accessing a privatized variable should not ideally increase when the variable is privatized. Per-access overheads can cause significant overheads if those variables are referenced in the innermost loops of computation. In order to validate that none of our methods exhibit this overhead, we ran a three dimensional Jacobi solver where all of the variables in the innermost computational loops are privatized. Figure 7 shows that indeed there are no hidden costs to accessing privatized variables compared to unprivatized variables. We have seen privatized variable access incur overheads with TLSglobals in the past, but we were not able to replicate it here. We hypothesize that any overhead can be optimized away by compilers when compiling with optimizations as we have.

4.4 Migration overhead

One of the main benefits of AMPI compared to traditional MPI libraries is its runtime support for dynamic load balancing, without the need for intrusive application code changes. The efficiency of dynamic load balancing depends in part on the cost of migrating ranks in AMPI. Since PIPglobals and FSglobals do not support

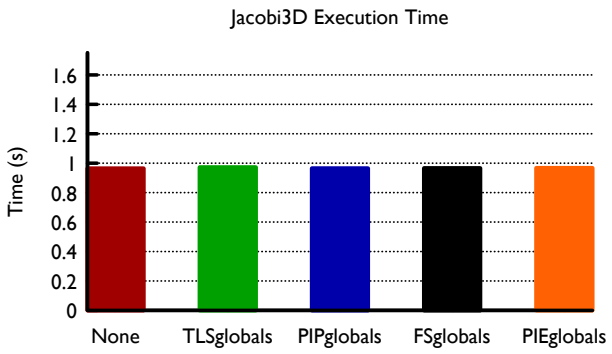

Figure 7: Execution time of Jacobi-3D where all variables accessed in the innermost loop are privatized global variables (lower is better).

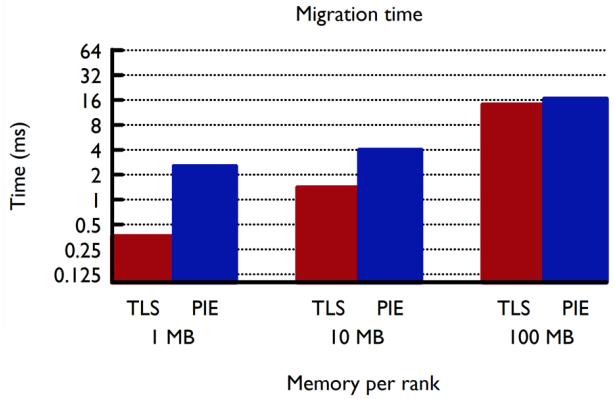

Figure 8: Migration time of virtual ranks with different sizes of memory allocated, comparing TLSglobals to PIEglobals (lower is better).

migration at all, they are not shown here. Ideally privatization would only increase the migration time proportionally to the cost of communicating the size of the privatized variables. With PIEglobals, however, we must communicate not only the globals themselves but the entire code segments as well. Of course the cost of migration then depends on the code size. Figure 8 shows results for ADCIRC, a production simulation code of approximately 50,000 source lines of code (described below), which has code size of approximately 14 MB that must be additionally migrated under PIEglobals but not TLSglobals. For reference, our Jacobi-3D standalone benchmark is around 100 lines of code and has a PIEglobals code segment size of 3 MB. Accordingly, as the (heap) memory per rank increases from 1 MB to 100 MB the proportional impact of PIEglobals on migration time decreases since the code segment consumes less of the rank's memory proportionally. This migration cost could potentially limit performance for fine-grained applications or when strong scaling with dynamic load balancing, but since dynamic rank migration is, in practice, relatively infrequent in applications using dynamic load balancing, we consider this cost high but acceptable. Furthermore, we discuss ideas for minimizing the migration cost in our future work section.

4.5 Instruction cache misses

Another concern we had when implementing our three new methods was that the code duplication would result in unnecessary instruction cache misses. This could potentially affect the performance of all code, not just privatized variables, and slow down the entire application's execution. We used the PAPI performance monitoring library [5] to track instruction cache misses, but found the results surprising: on Bridges2 PIEglobals had 22% fewer L1 instruction cache misses than TLSglobals on our a Jacobi-3D benchmark. This was unexpected, so we ran the same benchmark on TACC's Stampede2 supercomputer, using its Intel Xeon Ice Lake nodes, and there TLSglobals had 15% fewer L1 instruction cache misses. Consequently, we are unable to draw a strong conclusion from PAPI counters on the instruction cache behavior of PIEglobals at this time– more investigation is needed, although application results suggest there is no significant overhead here.

Table 2: ADCIRC speedup of best performing virtualization ratio over the baseline (without virtualization or load balancing).

Cores	1	2	4	8	16	32	64
Speedup %	13	59	79	70	43	24	17

4.6 Application demonstration: ADCIRC

Lastly, we looked at overall execution time of a production application on PIEglobals. Demonstrating PIEglobals on a full-scale application code is important because the size of the code segments can increase the memory footprint and migration times as we have seen. In order to validate the technique, it must be applicable to large legacy codebases. Of our three novel methods, we only consider PIEglobals production-worthy because of its support for dynamic rank migration.

ADCIRC is a Fortran90 MPI code used to simulate storm surge flooding in real-time. It is used by the US Army Corp of Engineers, the Federal Emergency Management Agency, and the National Oceanic and Atmospheric Administration to predict the surge of rising ocean waters over floodplains, through low-lying marshes, and into communities during natural disasters such as hurricanes [19].

The ADCIRC code base originally contained hundred of mutable global variables across nearly 50,000 source lines of code. Privatizing all global state manually would be cumbersome, and the code is used on many different systems by users of varying degrees of HPC expertise, meaning requiring modifications to compilers or other system components would be burdensome to maintain and package. PIEglobals addresses these concerns with its portability and ease of use.

To validate PIEglobal's performance, we ran ADCIRC on Bridges2 with varying degrees of virtualization and with load balancing. Dynamic load balancing is particularly effective for ADCIRC since the computationally intensive parts of the domain follow the flow of water as it spreads over and around obstacles in its path. For dry

Table 3: Summary of privatization methods and their features, including our three novel runtime methods.

Method	Automation	Portability	SMP Mode Support	Migration Support
Manual refactoring	Poor	Good	Yes	Yes
Photran	Fortran-specific	Good	Yes	Yes
Swapglobals	No static vars	Linker-specific	No	Yes
TLSglobals	Mediocre	Compiler-specific	Yes	Yes
-fmpc-privatize	Good	Compiler-specific	Yes	Not implemented, but possible
PIPglobals	Good	Requires GNU libc extension	Limited w/o patched glibc	No
FSglobals	Good	Shared file system needed	Yes	No
PIEglobals	Good	Implemented w/ GNU libc extension	Yes	Yes

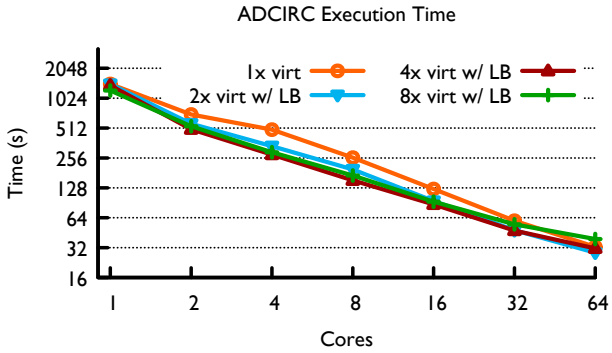

Figure 9: Strong scaling execution time for ADCIRC with varying degrees of virtualization and with dynamic load balancing (lower is better).

areas, there is little to no computational work. We use the built-in GreedyRefineLB load rebalancing strategy.

We note that PIEglobals successfully privatizes this large code base with hundreds of global variables, and it does so efficiently and portably while supporting dynamic rank migration. The overall result is that ADCIRC is able to perform between 79% and 13% better than the baseline without virtualization and load balancing thanks to PIEglobals. Even at the limits of strong scaling where communication tends to dominate performance we see a 17% improvement, and we expect more tuning of load balancing frequency and strategy can yield greater speedups as well.

5 RELATED WORK

We discussed the most directly related prior work at length in Section 2. To summarize, existing methods have taken varied approaches such as source-to-source refactoring tools [16], compiler extensions [4], runtime TLS segment switching [23], hierarchical extensions to TLS [21], and runtime ELF Global Offset Table switching [23]. We took particular inspiration from the Process-in-Process library and its runtime techniques for shared address space programming [11]. Our PIEglobals method improves on PIP's portability and support for virtual rank migrartion, while avoiding the need to patch glibc for SMP mode support or high virtualization ratios.

Other MPI runtimes that have been developed based on threads rather than processes include TMPI [20], FG-MPI [15], and Habanero-C MPI [7]. The MPI endpoints proposal [8] seeks to achieve a similar communication model as these except in a standardized approach that gives users the ability to choose the number of endpoints without requiring full-on process virtualization. Instead the user must manage memory more carefully, since the change in execution model is not transparent to legacy codes. Similar to the MPI endpoints proposal, OpenSHMEM contexts have been proposed to enable increased communication concurrency and overlap [9]. Process virtualization could potentially be applied to OpenSHMEM and other parallel programming models as well.

6 CONCLUSION

With the emergence of exascale class systems and cloud computing platforms, HPC application developers are facing a variety of challenges in evolving their codes forward to new levels of performance and simulation capabilities, all while ensuring correctness and maintainability. At the same time, task-based programming models are growing in appeal with their automated scheduling capabilities, asynchronous data movement support, and dynamic resource management features. However, since the investment in production software is large– often sustained over decades– the prospect of rewriting it for a new programming model can be daunting. Virtualization of existing codes is one approach to facilitate this transition, with fully automatic privatization being the ideal method of accomplishing it.

In this work we summarized the current state of the art approaches to code privatization and discussed the advantages and limitations of each, before detailing our three new runtime methods and evaluating them for performance. We believe that one of our new methods, PIEglobals, is the best privatization method developed yet in terms of portability across different architectures and compilers, applicability to both C/C++ and Fortran codes, runtime performance, and support for runtime migration of virtualized entities. It enables running legacy applications that we could not practically virtualize before on top of AMPI for its dynamic runtime support. Of course, for a particular runtime, application, and execution environment the importance of these criteria will be weighed differently. We place particular importance on the degree of automation, the amount of developer effort and expertise needed to apply it, support for migratability, portability across popular systems, and performance, especially minimizing context switch overhead and privatized variable access costs.

For future work, we plan on continuing to validate and test PIEglobals against production application codebases. We plan on exploring the use of dynamic binary instrumentation tools for scanning heap allocated static objects at startup for pointers that need updating to the privatized code segment. We also plan on adding support for Mac OS and on investigating memory optimizations. In particular, we are looking into reducing the code bloat issue of memory usage in PIEglobals by mapping the code segments into virtual memory from a single file descriptor using *mmap*. Further, we could potentially reduce its migration memory overhead by changing Isomalloc to only migrate segments of code that differ across different ranks. Having a way to detect read-only global variables and not duplicate them could reduce memory footprint per-rank as well.

ACKNOWLEDGMENTS

We would like to thank the NSF PREEVENTS grant ICER-1855096 and our collaborators there for supporting this research.

REFERENCES

[1] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. 2014. Parallel Programming with Migratable Objects: Charm++ in Practice *(SC)*.

[2] Gabriel Antoniu, Luc Bouge, and Raymond Namyst. 1999. An efficient and transparent thread migration scheme in the PM^2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) San Juan, Puerto Rico. Lecture Notes in Computer Science 1586*. Springer-Verlag, 496–510.

[3] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 66.

[4] Jean-Baptiste Besnard, Julien Adam, Sameer Shende, Marc Pérache, Patrick Carribault, Julien Jaeger, and Allen D. Maloney. 2016. Introducing Task-Containers as an Alternative to Runtime-Stacking. In *Proceedings of the 23rd European MPI Users' Group Meeting* (Edinburgh, United Kingdom) *(EuroMPI 2016)*. Association for Computing Machinery, New York, NY, USA, 51–63. https://doi.org/10.1145/2966884.2966910

[5] Shirley Browne, Christine Deane, George Ho, and Philip Mucci. 1999. PAPI: A Portable Interface to Hardware Performance Counters.

[6] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21 (August 2007), 291–312. Issue 3. https://doi.org/10.1177/1094342007078442

[7] Sanjay Chatterjee, Sagnak Tasırlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. 2013. Integrating Asynchronous Task Parallelism with MPI. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 712–725. https://doi.org/10.1109/IPDPS.2013.78

[8] James Dinan, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. 2013. Enabling MPI Interoperability through Flexible Communication Endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting* (Madrid, Spain) *(EuroMPI '13)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/2488551.2488553

[9] James Dinan and Mario Flajslik. 2014. Contexts: A Mechanism for High Throughput Communication in OpenSHMEM. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* (Eugene, OR, USA) *(PGAS '14)*. Association for Computing Machinery, New York, NY, USA, Article 10, 9 pages. https://doi.org/10.1145/2676870.2676872

[10] Message Passing Interface Forum. 2015. *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart. https://books.google.com/books?id=Fbv7jwEACAAJ

[11] Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jai Dayal, Pavan Balaji, and Yutaka Ishikawa. 2018. Process-in-process: Techniques for Practical Address-space Sharing. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (Tempe, Arizona) *(HPDC '18)*. ACM, New York, NY, USA, 131–143. https://doi.org/10.1145/3208040.3208045

[12] Chao Huang, Orion Lawlor, and L. V. Kalé. 2003. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*. College Station, Texas, 306–322.

[13] Nikhil Jain, Abhinav Bhatele, Jae-Seung Yeom, Mark F. Adams, Francesco Miniati, Chao Mei, and Laxmikant V. Kale. 2015. Charm++ & MPI: Combining the Best of Both Worlds. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (to appear) (IPDPS '15)*. IEEE Computer Society. LLNL-CONF-663041.

[14] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 6.

[15] Humaira Kamal and Alan Wagner. 2010. FG-MPI: Fine-grain MPI for multicore and clusters. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 1–8. https://doi.org/10.1109/IPDPSW.2010.5470773

[16] Stas Negara, Gengbin Zheng, Kuo-Chuan Pan, Natasha Negara, Ralph E. Johnson, Laxmikant V. Kalé, and Paul M. Ricker. 2010. Automatic MPI to AMPI Program Transformation Using Photran. In *Proceedings of the 2010 Conference on Parallel Processing* (Ischia, Italy) *(Euro-Par 2010)*. Springer-Verlag, Berlin, Heidelberg, 531–539.

[17] Stas Negara, Gengbin Zheng, Kuo-Chuan Pan, Natasha Negara, Ralph E. Johnson, Laxmikant V. Kale, and Paul M. Ricker. 2010. Automatic MPI to AMPI Program Transformation using Photran. In *3rd Workshop on Productivity and Performance (PROPER 2010)*. Ischia/Naples/Italy.

[18] Marc Perache, Herve Jourdren, and Raymond Namyst. 2008. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing* (Las Palmas de Gran Canaria, Spain) *(Euro-Par '08)*. Springer-Verlag, Berlin, Heidelberg, 78–88. https://doi.org/10.1007/978-3-540-85451-7_9

[19] KJ Roberts, JC Dietrich, D Wirasaet, WJ Pringle, and JJ Westerink. 2021. Dynamic load balancing for predictions of storm surge and coastal flooding. *Environmental Modelling and Software* 140, 105045. https://doi.org/10.1016/j.envsoft.2021.105045

[20] Hong Tang, Kai Shen, and Tao Yang. 1999. Compile/Run-Time Support for Threaded MPI Execution on Multiprogrammed Shared Memory Machines. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Atlanta, Georgia, USA) *(PPoPP '99)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/301104.301114

[21] Marc Tchiboukdjian, Patrick Carribault, and Marc Pérache. 2012. Hierarchical Local Storage: Exploiting Flexible User-Data Sharing Between MPI Tasks. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 366–377. https://doi.org/10.1109/IPDPS.2012.42

[22] Sam White and Laxmikant V. Kale. 2018. Optimizing point-to-point communication between adaptive MPI endpoints in shared memory. *Concurrency and Computation: Practice and Experience* (2018), n/a–n/a. https://doi.org/10.1002/cpe.4467

[23] Gengbin Zheng, Stas Negara, Celso L. Mendes, Eduardo R. Rodrigues, and Laxmikant V. Kale. 2011. Automatic Handling of Global Variables for Multithreaded MPI Programs. In *Proceedings of the 16th International Conference on Parallel and Distributed Systems (ICPADS) 2011*.