Introducing Task-Containers as an Alternative to Runtime-Stacking

Jean-Baptiste Besnard
ParaTools, SAS
Bruyéres-le-Châtel, France
jbbesnard@paratools.fr

Julien Adam
ParaTools, SAS
Bruyéres-le-Châtel, France
adamj@paratools.fr

Sameer Shende
ParaTools Inc.
Eugene, USA
sameer@paratools.com

Marc Pérache
CEA, DAM, DIF
F91297 Arpajon, France
marc.perache@cea.fr

Patrick Carribault
CEA, DAM, DIF
F91297 Arpajon, France
patrick.carribault@cea.fr

Julien Jaeger
CEA, DAM, DIF
F91297 Arpajon, France
julien.jaeger@cea.fr

ABSTRACT

The advent of many-core architectures poses new challenges to the MPI programming model which has been designed for distributed memory message passing. It is now clear that MPI will have to evolve in order to exploit shared-memory parallelism, either by collaborating with other programming models (MPI+X) or by introducing new shared-memory approaches. This paper considers extensions to C and C++ to make it possible for MPI Processes to run into threads. More generally, a thread-local storage (TLS) library is developed to simplify the collocation of arbitrary tasks and services in a shared-memory context called a task-container. The paper discusses how such containers simplify model and service mixing at the OS process level, eventually easing the collocation of arbitrary tasks with MPI processes in a runtime agnostic fashion, opening alternatives to runtime stacking.

Keywords

MPI+X; Thread-Based MPI; Privatization; In-Situ

1. INTRODUCTION

The continuously increasing number of cores inside computing nodes is raising questions concerning the evolution of the Message-Passing Interface (MPI) programming model. The MPI model is the de facto standard for the last two decades, strongly shaping how supercomputers are being programmed with a distributed memory paradigm. However, the notion of programming hundreds of cores in a physically shared memory node with a message passing paradigm using hundreds of MPI processes is unsettling at several levels. First, binaries and shared objects are loaded hundreds of times, leading to extra pressure on the batch manager and on the file-system. Second, the MPI communication layer has to be initialized for each of these processes. Depending on the communication topology and networking technology, multiple buffers and queues for each process must be configured. Furthermore, it can be easily shown that common patterns of communications (e.g., stencils with spatial dependencies) necessarily causes data and computing replication when a distributed memory context[2] is applied, further forcing larger memory areas for efficient implementation.

These observations have led to MPI + X paradigms, where MPI's inter-node distributed memory model and X's intra-node shared memory model come together to exploit multiple levels of parallelism[9]. Clearly, MPI is robust and necessary for communication between nodes. It makes little sense to replace it. However, X a shared memory model is generally less defined, though several candidates are found, including PGAS[6], OpenMP[7], TBB[19], and accelerators[25, 28]. In general, the purpose is to combine programming methods that can transition MPI-only program to models that can express the parallelism inside many-core nodes effectively, while addressing memory and replication issues.

Our work takes a fresh look at this debate, one that questions the boundaries between OS processes, MPI Processes and collocated processing in general for HPC applications. In fact, we propose a new level of abstraction between processes and threads, one that we called the *task* level, with the purpose being to ease the expression of collocated computation. We then offer a compiler-based transformation approach to generate MPMD executables based on task-level support. In our opinion, HPC workloads are evolving from a stacked paradigm to a horizontal one, eventually dedicating resources to analysis and data-management phases which were previously addressed by a subset of the machine (often called a *service island*). This will lead to an increased variability of the HPC payloads, not only embedding MPI and its associated shared-memory model X, but also other commodity services. For example, the transitioning of IO models to In-Situ[10] operations with burst-buffers and active buffers [21] could benefit from a richer programming methodology. Similarly, performance and debugging tools which utilize runtime support (e.g., tree-based overlay networks[14, 1]) or dedicated resources [3] could also be created more directly.

Our work develops new language abstractions enabling the use of the well-known MPI paradigm in a heterogeneous context mixing arbitrary collaborating computations and ser-

EuroMPI '16, September 25-28, 2016, Edinburgh, United Kingdom

DOI: http://dx.doi.org/10.1145/2966884.2966910

vices. The contribution of this paper is to provide data manipulation primitives for defining arbitrary contexts for tasks running in a common OS process. With such support, the transparent multiplexing of OS process-based payloads inside the same shared-memory context is possible. The paper first describes the *task-container* abstraction and its actuation in an MPI context in Section 2. After describing related work, we then discuss in Section 4 the internals of our implementation and the components enabling *task-containers*: the compiler, the linker, and a dedicated runtime library. Next in Section 5, we illustrate the use of our abstraction with user-level threads in the context of the MPC thread-based MPI. The paper concludes with thoughts for future work.

2. TASK CONTAINERS

The overall architecture we envision for the *task-level container* pattern is described in this section, along with the C language extensions we propose to enable such behavior in a regular program using the Executable and Linkable Format (ELF).

2.1 Overview

Consider the example of an IO library in the context of two MPI processes with remote burst buffers. This configuration could be illustrated as shown in Figure 1.

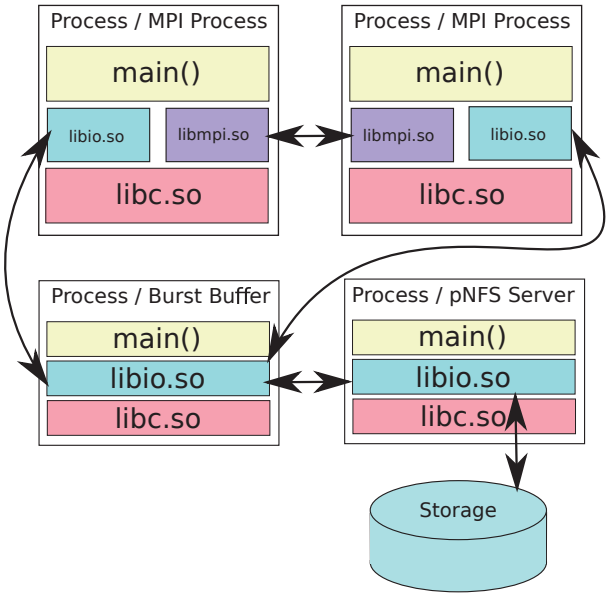

Figure 1: MPI Program mixed with a parallel IO workflow including a Burst-Buffer stage.

Here we present a generic IO work-flow in a process-based MPI context. Each process is linked to an IO library implementing the IO transport layer interfacing with burst buffers. Similarly, each process is linked to the MPI library which furnishes all the functionality enabling the process to become an MPI process involved in a distributed memory computation. Thus, processes are interacting through two communication layers: the MPI layer and the IO layer. Notice that the IO library is duplicated for each process and incurs computation there, an approach that we describe as stacking. One could argue that the MPI process in the MPI + X model could occupy the complete node, as a consequence, the IO library would be present only once per node, solving this duplication issue for IO and MPI. However, this splitting does not take the computational cost into account. Indeed, simple tasks such as moving data around require computing power, as acknowledged by burst/staging buffer approaches aimed at hiding such IO costs.

Consequently, the question we address in this paper is how multiple collaborating processes can be expressed in a shared-memory context, not only exposing features through an API consuming local computing cycles (stacking), but also providing such processing with dedicated computing resources while remaining accessible through regular API calls (horizontal/In-Situ). This involves not only the mixing of various API (linking to libraries), but also the expression of multiple processing or *tasks* in the same program running on a many-core node. Figure 2 shows how this model might operate for the scenario in Figure 1.

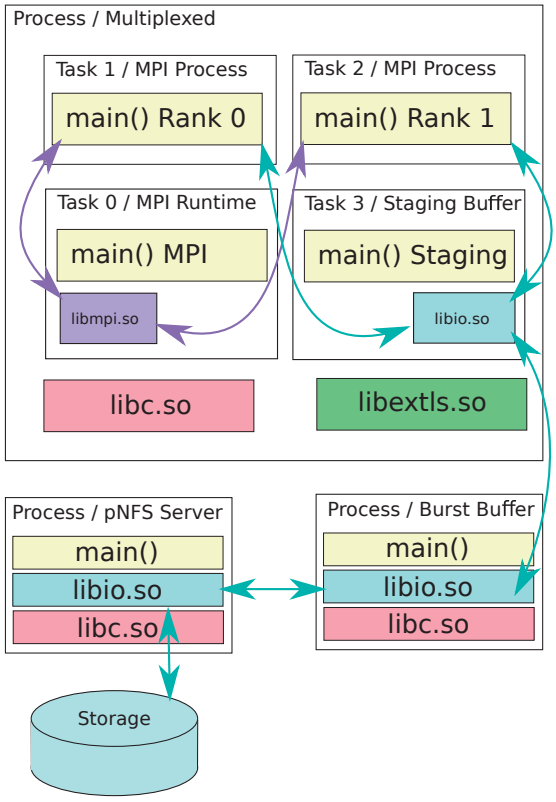

Figure 2: Multiplexed MPMD MPI program with a parallel IO workflow including a Burst-Buffer stage.

Figure 2 uses task-level data-sharing to enable multiple tasks to run in the same OS Process. It not only instantiates each library only once, but can dedicate processing power to each of them. In this example, the IO services are running inside the same shared-memory space as multiple MPI Processes, allowing direct interaction through API calls while explicitly dedicating resources to each of the tasks. Task-containers can be seen as running different main functions on different cores, mimicking what was present at process level in Figure 1 inside task-containers multiplexed inside a

single OS Process. In fact, this model transposes the advantages of a thread-based MPI and alternative definitions of MPI processes such as endpoints[8] or MPI sessions to existing libraries requiring both a context and processing power. Not only does this model circumvent data-replications associated with the message-passing model in distributed memory context (e.g., by efficiently using ownership passing messages[13]), but it also opens wider perspectives. For instance, Figure 2 shows how some core might be dedicated to the MPI library in order to progress MPI communications, creating a task-container dedicated to the MPI runtime while improving polling behavior [15].

More generally, any service which may need to be collocated with multiple MPI processes could be isolated in a task-container. In this way, non-MPI MPMD services could be linked to the MPI applications. Using this model, any IO, runtime library, or tool could be made adjunct to running processes through an automatic compiler transformation. The advantage is that the method is applicable to MPI without being inside MPI. Indeed, we started this paper on the observation that model mixing was a compulsory avenue for many-core. However, one of the issues in this approach is that it is generally MPI which does resource negotiation, based on OS process granularity. Our work introduces a layered data-management technique through compilation primitives to enable transparent and backward compatible data-sharing constructs, a point that is the main contribution of this paper.

Our proposal is to extend the thread-local storage (TLS) model to integrate a new level between threads and global variables. This extended TLS model allows teams of threads (or tasks) even entire process contexts (all the global variables at all levels) to be duplicated inside a single OS Process. In such a model, each library could specify through new C/C++ keywords its own context relative to global variables, indeed having its internal context spanning global variables at different levels. Our work aims to extend the global C/C++ variable semantic to express collocated processing at compiling/linking time.

2.2 Extended TLS Handling

In order to obtain the behavior of Figure 2 in a transparent manner, we decided to rely on compiler-based transformations. There are multiple requirements. First, the approach used must be unintrusive and compatible with legacy codes. Second, the approach should be model-independent, although our original target was the MPI model. Relying on a low-level mechanism such as TLS makes this possible.

More practically, a new logical TLS level called the *task* level is introduced. C/C++ programmers, already commonly use the `__thread` keyword which makes a global variable thread-local. Similarly, we introduce the `__task` keyword which is an intermediate between a process-level global variable and a thread-level one. Thus, this level targets a team of threads. Moreover, as we wanted this transformation to be transparent for existing programs, a choice is made of transitioning global variables to task-local variables, adding the need for a `__process` keyword to prevent automatic transition.

As shown in Figure 3, the task-level outlines teams of threads while separating them from the global process context. It is then an intermediate between no-sharing (`__thread`) and complete sharing (`__process`). On the programming

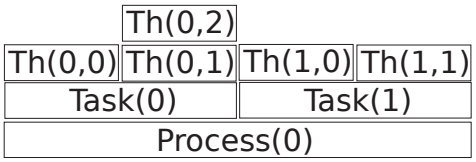

Figure 3: Context stacking using task-containers.

side, issuing data in each of these levels can be done for C in a straightforward manner as listed in Figure 4.

```
1 __thread int thread_id;
2 __task int task_id;
3 __process int process_id;
```

Figure 4: Using exTLS levels in C

The simple syntax of Figure 4 is a key element in our task-container model. Indeed, as we are working at compilation level, altering TLS, it is possible to compose the behavior of multiple libraries present for example in shared-objects. This allows, for example, part of an MPI library context to be located at the `__process` level and therefore shared between all tasks and threads. Similarly, if some context has to be local to a task (which in our vision is matching the MPI Process level), one can create variables at `__task` level to store its local context, and it will be inherited by all the siblings threads. More generally, this model can be applied outside of MPI, with an arbitrary processing partially sand-boxed. Indeed, these keywords allow libraries to collaborate through their programming interface without completely sharing the same data-context thanks to an additional logical TLS data-hierarchy.

One point that we omitted for now is how contexts are inherited. Process and thread context can be seen as semantically linked to an execution level, but in fact (and as we will further describe), they should be regarded in a more flexible manner. In next Section, we introduce the *inheritance* mechanism to define how data-sharing level are manipulated between tasks.

2.3 Context Inheritance

Context inheritance is the mechanism allowing the definition of sub-contexts in the OS Process. Given that any execution stream has TLS data for each level, inheritance is the way of subdividing these levels between collocated tasks in an arbitrary manner.

```
1 int inherit_thread();
2 int inherit_task();
3 int inherit_process();
```

Figure 5: Context inheritance interface.

The function calls presented in Figure 5 are the primitives we introduce to subdivide TLS contexts in a multi-threaded application, eventually creating tasks and as we will see process containers. We now propose to consider the example of an MPI application.

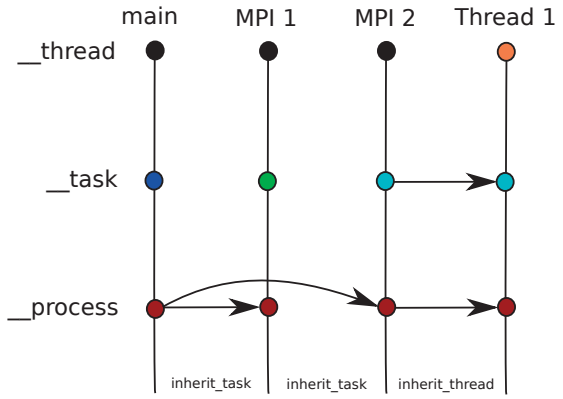

Figure 6: Spawning two thread-multiple MPI Processes in a task-container context.

As presented in Figure 6, in our model, two MPI Processes can be spawned by an MPI library just by calling the `inherit_task` call. This will have the effect of duplicating TLS levels superior (see Figure 6) to the *process* one eventually creating a task context. Similarly, if we look at the MPI Process 2, if it spawns a thread, it has to run in the same task context, but with a different thread-level set of variables. To do so, the `inherit_thread` function is invoked. This data-management approach is then sufficient to define a thread-based MPI where the task level is associated with the MPI process one, and the data-sharing level is defined explicitly by the host runtime via the *inheritance interface*.

To go further, we can illustrate the versatility of our approach through model-mixing, launching multiple MPI processes sharing the same OpenMP runtime. We simply consider that the OpenMP runtime is compiled at process-level, and therefore exposes a shared context between tasks. With our task-container model, the OpenMP runtime would then have to initialize its own task context, before spawning its worker threads on dedicated resources. This would lead to multiple sequential MPI tasks sharing the same OpenMP workers.

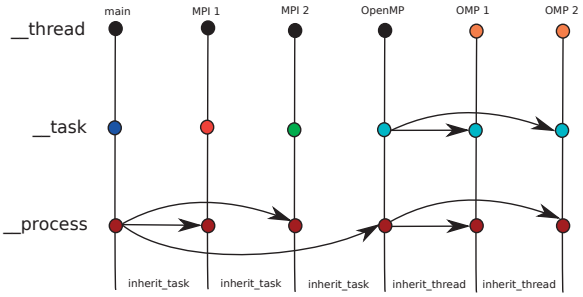

Figure 7: Two MPI Processes sharing the same OpenMP runtime.

The idea behind the hybrid-mixing model illustrated in Figure 7 is that the OpenMP library context has not been *privatized*. Therefore, unlike MPI processes, it is exposed through the OpenMP ABI to the whole OS process. Consequently, parallel regions invoked by the MPI Processes will be scheduled on the threads of the OpenMP task which has no sibling relationship with the MPI Processes. Our extended TLS model then allows an OpenMP team to be shared between multiple MPI task instead of relying on the classical nested MPI+X relationship where OpenMP runs inside MPI.

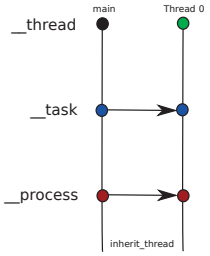

Figure 8: Illustration of the `inherit_thread` call.

One point that is important to understand in this new model is that each execution stream has a copy of all variables and therefore is by itself fully defined in all levels. This is the reason why the "bars" of Figures 6 and 7 are going through all the levels. Consequently, if the *main* function emits an `inherit_thread` call prior to an `inherit_task` call, the new context will have a new thread segment while sharing both task and process levels with the original main stream as illustrated in Figure 8. In contrast, the `OMP 1` thread of Figure 7 inherited its context from the task associated with the OpenMP runtime.

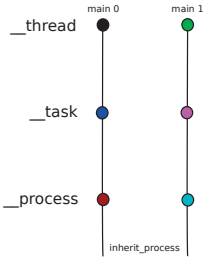

Figure 9: Illustration of the `inherit_process` call.

Besides, as depicted in Figure 9, inheriting from a process is similar to recreating a new TLS context for all levels, in fact mimicking what is happening when creating a normal OS process. It allows disjoint TLS containers to run in the same OS process. For instance, a process-based MPI could be run inside a single OS Process without changing its semantic through a simple recompilation.

To summarize, an inheritance call for a given level duplicates all TLS levels strictly inferior to the target level while creating new segments for upper levels. In complement, this call affects the current execution stream which has its TLS level modified and as a consequence, all the variables affected by level changes are modified. This simple extension of the TLS model with a few function calls is then a way to group threads running in a single OS Process inside task-containers. As we will illustrate in the next Section, our model opens new interactions between collocated tasks which would not have been possible between OS Processes.

2.4 Context Swapping

Figure 7 also covers an interesting facility of the extended TLS library, *context swapping*. Indeed, an MPI Process entering a parallel region may rightfully emit MPI calls depending on the MPI context. However, the two runtimes would have to negotiate in order to manipulate their respective TLS contexts (OpenMP and MPI). To do so, we introduce a *disguise* function call allowing a given stream to borrow another stream's context, while preserving its original context in a *shadow* context. In our OpenMP example, if the MPI process is willing to enter an OpenMP parallel region, it may pass its current TLS context as an extra parameter so that the OpenMP runtime *disguises* all its threads in MPI Processes before entering the parallel region. However, a parallel region can also emit OMP related calls such as `omp_get_thread_num`, obviously relying on the OpenMP context associated with the execution stream. This problem is addressed with the *shadow context*.

```
1  int omp_get_thread_num()
2  {
3    int ret = -1;
4    /* Here the stream is MPI disguised */
5    shadow_begin();
6    /* Here the stream is OpenMP */
7    ret = omp_thread_info->id;
8
9    shadow_end();
10   /* Here the stream is MPI disguised */
11   return ret;
12 }
```

Figure 10: Implementation of `omp_get_thread_num` relying on shadow contexts.

As outlined in Figure 10, a thread which is coming from another context is still able to query its own context by outlining shadow code regions, with the effect of restoring the original thread-context. In this example, OMP related descriptors can be accessed despite being inside a code region invoked from a foreign context. In complement, in order to allow the "normal" execution patter of OMP parallel region without borrowed contexts, the shadow context is aliased to the current context when the thread is not disguised. This allows the same code to address two different scenarios.

As a consequence, the task-level container is a convenient manner of expressing model-mixing without requiring a sibling relationship between models. In contrast to current MPI + X models where MPI launches processes, here models are collaborating in a horizontal manner. Therefore, any task such BLAS or IO library located in the same OS process may invoke the shared OpenMP library even if it is not MPI aware. To further illustrate context swapping, this mechanism also allows an IO thread to attach to running MPI processes in order to extract their local data from their global variables, without having to rely on a translation API. The MPI process could then provide what could be described as an IO write *lambda-context function* which would then be invoked by another thread with a shadow IO context as a convenient In-Situ abstraction. More generally, using this lambda-context semantic, polling threads could be shared between multiple tasks instead of being duplicated for each of them. To provide an MPI example, this would allow generalized requests[20] to be polled by any threads borrowing the MPI context when calling the progress handler.

3. RELATED WORK

Thread-based MPI implementations have been an active subject of research as a possible solution to many-core challenges. Our work is based on the MPC thread-based MPI [23] + OpenMP [4] runtime, which pursued the automatic compiler privatization model in order to transpose existing MPI codes to the thread-based MPI model. This privatization capability has been realized in the Intel ICC compiler through the `-fmpc-privatize` option[17]. MPC also investigated the use of TLS extensions relative to both topological constraints with Hierarchical Local Storage (HLS)[26] and OpenMP context with a dedicated TLS level [4]. The work done in this paper is thus an extension of this initial implementation which was intended for a thread-based MPI to a wider context. Moreover, our work aims at providing these features inside an open-source package to promote its wider usage. There are, of course, other thread-based MPI implementations that faced the same shared global variable issues. The propos solutions can be gathered in three categories: source-to-source, compile/linking time and runtime.

AMPI originally proposed solution relying on Photran[22] to gather global variables inside a module before modifying function invocation. This source-to-source privatization approach may be difficult to apply with C++ (or even C) due to more complex data-types and potential indirect references, which would require an extended data-flow analysis to transform the source-code. Another source-to-source transformation for global variables privatization is array expansion. It is often used at the compiler level to circumvent memory dependencies [27], replacing each variable with an array, accesses being indexed with the thread identifier. One aspect making this transformation very impractical is that the array has to be extended each time a thread is created (without moving it in the address space) to preserve potential pointer references. It makes this model potentially expensive at runtime, a point that adds itself to the inherent complexity associated with the parsing of language constructs (particularly in C++). Consequently, despite its natural portability, being at language level, the source-to-source approach poses several issues when it comes to privatizing existing codes due to the parsing it supposes. Moreover, it implies that the code and all its libraries have to be recompiled in order to be executable in a thread context.

Alternatively, a Clang compiler-based transformation for C and C++[24] can simply remove a global variable through by passing every global variable as a parameter to each function. One limitation of this model is that it does not handle aliasing – potentially missing some global variables. Moreover, as a compiler only compiles a single source file at a time, this transformation is not able to account for global variables present in other translation units (or worse in shared-libraries linked later on), making it unsuitable for larger code-bases. Similar to what was done in MPC at the gimple level for C, C++ and Fortran, the AMPI team investigated compiler-level privatization promoting global variables to the TLS level[29]. However, they did not cover the user-level thread case which also requires a custom TLS runtime library to perform manual context-switching. This poses the question of handling thread-local variables. Indeed, promoted global variables have to be shared between the threads belonging to a given MPI Process (now running inside a thread), making runtime support compulsory. Compiler based privatization can take advantage of the parsing

and analysis facilities provided by production-grade compilers to extract and promote global variables to TLS. However, TLS do not have the same semantics as global variables, particularly as they have a non-constant address, preventing them from being used to initialize global variables (pattern allowed with static variables in C[18], illustrated in Figure 15). This aspect has to our knowledge not being covered by any compiler-based privatization framework. In this paper we propose a solution to this issue, greatly enhancing the range of codes which can be privatized. Eventually, as for source-to-source approaches, compiler-based privatization requires the program and all its libraries to be recompiled. Moreover it supposes that the target architecture provides TLS support.

For runtime approaches, the GOT global privatization technique aims at context switching the Global Offset-Table (GOT) for each shared-object. However, the paper[29] is unclear about how GOT context-switching is achieved when considering multiple threads; it seems only applicable for co-routines which are sequentially executed. Moreover, the GOT approach is only feasible for global variables present in position independent sections of the code, static variables being accessed through direct register offsetting. Another runtime approach has been proposed by Hybrid MPI (HMPI)[12] which solved the privatization issue in a very reliable manner, simply by avoiding it. Indeed, Hybrid MPI relies on actual OS processes to separate ranks, but provides a modified allocator relying on a shared-memory segment. As a consequence, each MPI process has its own stack, its own variables and code section, but the heap is shared between processes. One limitation of HMPI is message content located on the stack which may not be in the shared segment and therefore not able to take advantage of the thread-based nature of the runtime. Moreover, allocating a shared-memory segment between processes necessarily maps physical pages, preventing the target application to take advantage of virtual memory. As a consequence, some codes oversubscribing memory may exhaust available shared-segment space which is bound to be a subset of physical memory. Nonetheless, runtime approaches have the strong advantage of avoiding any form of recompilation or patching of the target program, it makes them very convenient to an end-user's point of view, as a given program may depend on a wide range of libraries some of them being possibly installed system-wide.

To summarize, none of the global variables privatization techniques is fully satisfying. We retained the compiler-based transformation as it was the only one capable of transposing a normal process-based program to a task-container, while preserving the initial thread-local variable semantic, possibly shared between multiple threads. This opens the way to the transparent transposition of existing MPI codes to task-containers in order to be run in a collocated manner, a point which is crucial when considering the case of existing codes, libraries and tools.

4. IMPLEMENTATION

In order to implement the task-container model presented in Section 2, we had to develop an extended TLS (exTLS) library able to manipulate several levels of TLS. This work is based upon prior work on the MPC thread-based MPI runtime which embedded a privatizing compiler promoting global variables to MPI Process local variables. Our purpose when developing this new library was to start from this initial MPI dependent implementation while opening it to a wider usage and community by enriching it and the associated programming languages (C and C++) with the concepts it supposed abstractively: task-containers. After a recall of the TLS implementation in the libC, we are going to describe our extended TLS implementation. To do so, we will first describe our work at the compiler level on TLS wrapping and dynamic TLS initializer handling for C. Then, we describe the exTLS runtime, introducing a multi-level TLS data-structure. Eventually, we describe our modifications of the linker to preserve extended TLS support.

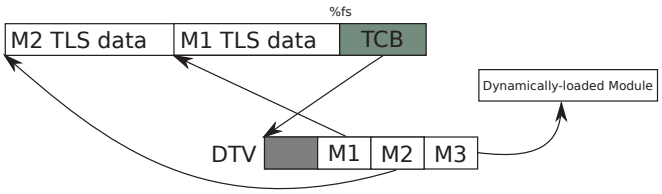

Figure 11: TLS data-structure in the GNU-libC.

4.1 TLS Support in the libC

In order to give some context to the developments of this paper we will first describe the initial TLS mechanism in the Linux operating system. This description does not intend to be exhaustive, only providing an introduction to the TLS mechanism for users who are unfamiliar with its implementation. The GNU-C library already supports a TLS mechanism at the thread level; this mechanism has been extensively described by Drepper[11], it relies on multiple collaborating components in order to provide a convenient thread-local variable abstraction. First the compiler has been modified to include a new keyword, `__thread` which informs the compiler that a given global variable shall be duplicated for all threads. Internally, the compiler will replace references to these variables with a function call `tls_get_addr` taking two parameters: a module and an offset. The module is associated to a compilation object, the binary or linked libraries, each of these modules can be seen as the aggregation of the `.tdata` and `.tbss` segments of the respective ELF objects. Moreover, all the modules are allocated in a contiguous segment in order to allow, as we will further describe, linker-level optimizations.

As shown in Figure 11, at runtime, each thread is given a TLS segment which is accessible directly using the `fs` register (for the x86-64 architecture that we are going to focus on). This register points to the start of *Thread Control Block* (TCB) containing, among other things, a pointer to the *Dynamic Thread Vector*. This DTV is, in practice, an indirection table translating a module ID to a base address in the contiguous TLS segment. Given these components, one can see how practically a module plus offset `tls_get_addr` call can be translated to a pointer in the TLS segment. As far as offsets and module identifiers are concerned, they are mostly handled by the linker during the relocation process, through `Global Offset Table` (GOT) entries filled by the loader[11].

The linker plays a very important role in TLS handling, first taking care of the relocation of the TLS inference entries but also more importantly optimizing TLS accesses depend-

```
// Resolve GOT entry (Module + Offset)
lea i@tlsgd(%rip),%rdi
// Call get_addr
call __tls_get_addr@plt
```

Figure 12: Global Dynamic (GD) TLS access.

ing on their context. In particular, there are four optimization levels implemented in the GNU linker, two involving runtime calls and two others relying on the register offsetting. In Figure 12, we present the assembly code associated with the default TLS access pattern described at the beginning of this section, it is the less optimized one. Note that this access model is at the basis of the second level of optimization which is Local Dynamic (LD). It simply consists in storing the base of the TLS offset with a function call before referencing module-local variables through direct offsetting relatively to this base pointer stored in a register.

```
// Direct offsetting of the %fs pointer
lea %fs:i@tpoff,%rax
```

Figure 13: Local Exec (LE) TLS access.

In contrast, if we look at Figure 13, we see the most optimized access model Local Exec, which is register-based. This optimization is only possible for TLS present in the main binary as the offset (`i@tpoff`) is statically resolved and written in the text-section of the binary. The reference register is `fs` and the offset is as shown in Figure 11, negative to address the static TLS segment. Eventually, register offsetting is also possible for shared-libraries the only difference with the LE mode is that the offset has to be resolved through a GOT inference filled by the loader which is in charge of mapping the static TLS segment at runtime. These optimizations are applied by the linker during the relocation process, for this reason our work on the Extended TLS model will also cover this software component.

The runtime library is also a crucial component in TLS handing. It first collaborates with the threading library which is in charge of switching the TLS register (`fs`), then it, of course, has to build the TLS context for each thread by concatenating all the static TLS segments in order while resolving related GOT entries (role of the loader). Moreover, the runtime has to handle dynamic symbol resolution for dynamically loaded object as presented in Figure 11, in such case, additional modules are added to the DTV and the corresponding TLS segment is dynamically allocated. In summary, TLS are covering the whole compilation chain, as a consequence, our task-level container implementation also had to spread on all these components.

4.2 Compiler-Level Extensions

We started our implementation with the MPC privatizing GCC compiler that promoted global variables to task-local variables. It provided us with the base for the addition of the `__process` and `__task` levels, the original MPC compiler only processed global variables without attributes and promoted them (transparently) to task-local variables. Dealing with thread-local variables, they were left thread-local (at user-level thread level). However, this initial approach encountered several limitations, particularly in C as we will describe. First, we added the `__process` and `__task` levels to the C front-end, defining new language keywords. Then, we defined new TLS levels which were already partially present in the MPC privatizing compiler. Eventually, we changed how references to these variables were emitted to match the function calls of Figure 14.

`__process`	`__extls_get_addr_process`
`__task`	`__extls_get_addr_task`
`__thread`	`__extls_get_addr_thread`

Figure 14: Substitute TLS calls associated with the new TLS levels

The addition of these new level of TLS ended up being relatively straightforward, it was done in a few hundred lines of code in the GCC compiler. Note that all these TLS levels are still managed as "normal" TLS in the sense that their relocation and insertion in the ELF sections are done as `__thread` TLSs. This saved us a lot of development time. Note that the `__process` level which is in fact matching non-privatized variables (normal globals) is also redirected to our exTLS runtime library. We made this choice to take full-control of all the global variables, allowing for example, the process-level duplication of Figure 9.

```
int a = 5;
int *b = &a;
```

Figure 15: Example of patterns posing problems when being privatized in C.

In complement of the insertion of these new TLS levels, we worked on the extension of the TLS support for dynamic initializers in C. Indeed, as shown in Figure 15 in C you can store an address to another global variable in a global variable. When privatized, such global variable is not constant anymore as by definition this global-variable address is now different for each thread (privatized). Transparent support for such patterns is crucial if we want to be able to compile an arbitrary C code to the task level, as the default compilation level in the exTLS compiler is *task*. Note that dealing with the C++ case, C++11 already provides an initial support for dynamic initializers for the `thread_local` keyword.

In order to enable the support for non-constant TLS initializers in C, we started with an analysis of the C++11 implementation associated with the `thread_local` keyword. This support relies on the systematic wrapping of TLS variables with function calls following the pattern described in Figure 16. The `_ZTW1aX` function returns a reference to the TLS after calling a `_ZTH1aX` function aliased to a per translation-unit `__tls_init` function with `X` the variable name. In the `__tls_init` function, all dynamic variables belonging to a given translation unit (TU, a preprocessed source-file) are initialized once thanks to the guard variable.

In C, we implemented a mechanism similar to what we found in C++11 to add support for dynamic TLS initalizers. This work has been done in a GCC plugin at the exception of some modifications to the GCC C front-end to allow non-constant TLS initalizers (at parsing time). As we

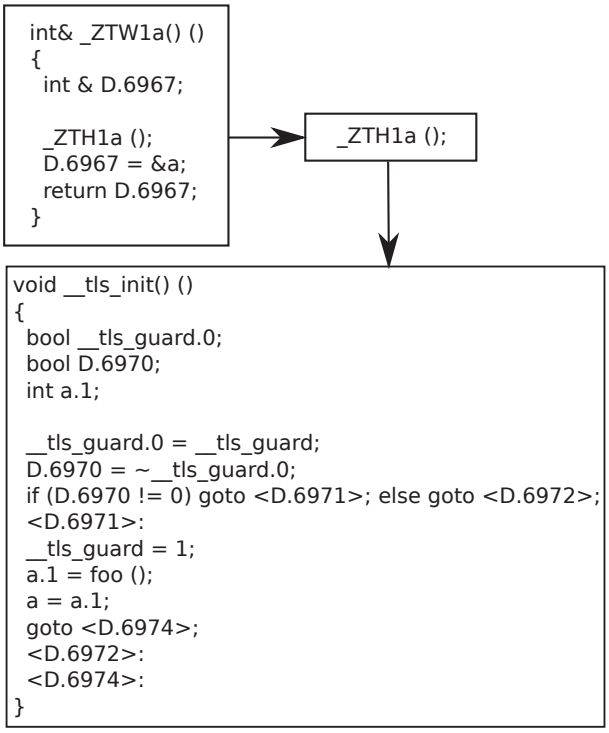

Figure 16: C++11 dynamic TLS wrapping for an integer variables `a` initialized by a function `foo()` (GCC Gimple output).

```
1  ___ex_TLS_w_b ()
2  {
3    tmp.3 = ex_tls_var_need_to_call_init.2;
4    if (tmp.3 == 1) goto <D.9940>; else goto <D↩
         .9941>;
5    <D.9941>:
6    ex_tls_var_need_to_call_init.2 = 1;
7    mpc_TLS_per_tu_init ();
8    <D.9940>:
9  }
10
11
12 ex_TLS_per_tu_init ()
13 {
14   tmp.1 =ex_tls_TU_guard.0;
15   if (tmp.1 == 0) goto <D.9934>; else goto <D↩
         .9935>;
16   <D.9934>:
17  ex_tls_TU_guard.0 = 1;
18   b = &a;
19   <D.9935>:
20 }
```

Figure 17: Gimple output for TLS initalizers in C using our plugin for the pattern of Figure 15.

```
1  extern int *b;
2  int **c = &b;
```

Figure 18: Translation-unit depending on the extern variable of Figure 15.

will further elaborate on, we avoided the systematic variable wrapping which has been retained by C++11, adding a second layer on top of the TLS resolution process and nullifying the advantage of TLS optimizations. As shown if Figure 17, we, however, retained the two level wrapping model with a wrapper function per variable and then a per-translation-unit initializer taking care of initializing all the non-constant TLS variables for the TU.

An important design point is how we handled data dependencies between TLS variables possibly scattered between multiple translation-units. For example, looking at Figure 18, if a variable `c` is a pointer to another dynamic variable in a remote translation unit, its initialization semantically requires the initialization of `b`. However, when compiling the translation-unit of Figure 18, we cannot infer if `b` is a dynamic TLS or not, or in other words, if the function `___ex_TLS_w_b` exists. We solved this issue by relying on the generated code of Figure 19. Extern TLS variables are systematically wrapped with local static getters calling the `ex_TLS_locate_dyn_initializer` function. This function simply does a `dlsym` of the target function, calling it if it does exist – following our example, propagating a data-dependency to the `b` of Figure 15. One of the advantages of relying on `dlsym` to query dynamic initializer functions is that the potential shadowing of global TLS variables present in multiple shared-libraries and then depending on linking order is correctly handled, transparently initializing the correct variable. Starting from this remote variable resolution, dependency propagation is implemented by calling the local wrapper for all extern TLS variables manipulated in the `ex_TLS_per_tu_init` function, in this example `b`. These calls are inserted at function start through an analyzing of the internal compiler representation. During this process, guards at both wrapper and per-translation-unit initializer function levels are crucial to prevent any circular initialization pattern, breaking recursive chains as the guard is always set before calling the remote initialization function. This relatively simple code is then sufficient to correctly initialize dynamic TLS between multiple translation-units while taking into account arbitrary dependencies between variables through prior invocation of potential remote initializers in the initialization function.

One point that we have not described yet is how local initalizers are invoked. We tried two different approaches. The first one consisted in patching at compilation time all functions depending on TLS, inserting for each function calls to the resolution wrapper for extern TLS (which cannot be inferred as non-dynamic) and the local wrappers for local TLS known to be dynamic. As such, each compiled function was first checking whether TLS it may depend on were initialized. This had the effect of adding several `if` tests at the beginning of each function, leading to a performance impact even if these tests were correclty inlined by the compiler. In order to mitigate this performance impact, we then developed a second approach relying on self-introspection and

```
1  ___ex_TLS_w_c ()
2  {
3    tmp.3 = ex_tls_var_need_to_call_init.2;
4    if (tmp.3 == 1) goto <D.10013>; else goto <D↩
         .10014>;
5    <D.10014>:
6    ex_tls_var_need_to_call_init.2 = 1;
7    mpc_TLS_per_tu_init ();
8    <D.10013>:
9  }
10
11
12 ___local_ex_TLS_w_b_6700841133086155300 ()
13 {
14   tmp.5 = ex_tls_var_need_to_call_init.4;
15   if (tmp.5 == 1) goto <D.10020>; else goto <D↩
         .10021>;
16   <D.10021>:
17   ex_tls_var_need_to_call_init.4 = 1;
18   ex_locate_tls_dyn_initializer ("___mpc_TLS_w_b↩
         ");
19   <D.10020>:
20 }
21
22 ex_TLS_per_tu_init ()
23 {
24   ___local_ex_TLS_w_b_6700841133086155300 ();
25   tmp.1 =ex_tls_TU_guard.0;
26   if (tmp.1 == 0) goto <D.10007>; else goto <D↩
         .10008>;
27   <D.10007>:
28  ex_tls_TU_guard.0 = 1;
29   c = &b;
30   <D.10008>:
31 }
```

Figure 19: Gimple output for TLS initalizers in C using our plugin for the pattern of Figure 18.

Wrapping Model	Per-function	Introspection
Walltime (seconds)	244,8	94,5

Figure 20: Privatized HDF5 library test-suite (testhdf5) sequential execution walltime in function of the dynamic TLS wrapping model.

therefore completely avoiding wrapping. As shown in Figure 17 and 19, TLS variables with dynamic initializers are associated with an `___ex_TLS_w_X` function, with `X` the variable name. Our alternative TLS initialization process relies on a self-scanning of the symbol table in the running binary and all its libraries in order to call all the dynamic TLS wrappers prior to launching the `main` program. To do so, TLS initializer functions are stored as a list of function pointers which are invoked each time a thread is created, correctly initializing dynamic TLS values prior to their execution.

As demonstrated by the timings of Figure 20, the introspection approach is much more efficient. We tested it by running the HDF5 sequential test-suite in a task-privatized mode (automatic privatization) using the two aforementioned wrapping techniques. This difference can be explained by the high modularity of the HDF5 package, indeed, it exposes its interface through global stuctures which are privatized. As a consequence, most HDF5 function do reference a TLS, requiring the addition of wrapping calls. In addition, HDF5 is a shared library and TLS calls cannot be optimized between functions of the library, increasing wrapper's overhead. In the light of this result, introspection is the default

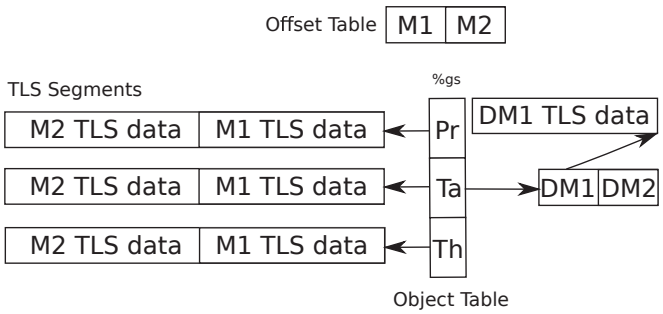

Figure 21: Data-structure of our exTLS library.

initialization model we retained. However, note that the per-function model is implemented and available through a compiler flag (-fmpc-dyn-insert). Note that the complete support of the introspection approach required the wrapping of the `dlopen` function calls in order to invoke TLS initalizers when loading dynamic objects. It also required a modification of the dynamic symbol table exports as we will discuss in Section 4.4. Our compiler support for dynamic TLS initalizer in C involves less function calls than the C++11 implementation which preferred lazy initializer invocation instead of our either program or function start model. We chose the direct initialization model as C forbids complex initializers, the only non-constant case when promoted to TLS being unresolvable references at compilation time (pointer to TLS). Consequently, calling all initializers at program start in C is not associated with potential memory or computation overhead unlike C++ which might for example initialize complex objects.

4.3 Extended TLS Library

In the previous Section, we explained how we added multiple TLS levels by defining new `get_addr` function calls matching task and process levels. Then, we detailed how we extended the C language support in GCC to allow dynamic TLS initializers to handle pathological privatization cases appearing with pointers to global variables which are correct C initializers invalid for privatized variables. In this section we are going to describe the runtime support associated with task-containers. To do so, we are first going to comment the TLS data-structure we retained, contrasting it with the one of the libC, presented in Section 4.1. Then, after explaining the TLS resolution process, we will comment edge cases such as dynamically loaded objects.

Figure 21 presents the data-structure of our exTLS library, structure which differs from the one of the libC (see figure 11) on several points. First, the Dynamic Thread Vector (DTV) storing a pointer to the start of each module in the TLS segment has been removed. We replaced it with a global (process-level) *offset table* containing the negative offset from the end of the static TLS segment, value which is thread-independent. In complement, we added a new table, the *object table*, it is associated with the TLS level from `__process` (Pr) to `__thread` (Th). This table is simply a pointer array with a pointer to the static TLS segment for each level. Note how we preserved the libC layout by pointing to the segment the same way the loader handles them, the purpose of this approach is to stay compliant with the

relocation process for GOT entries and offset calculations – greatly simplifying our TLS implementation. Each static TLS segment is mapped in the exact same manner as what is done for the libC, concatenating the `.tdata` and allocating a zero-filled `.tbss` section for each module. To ensure relocation compliancy, our implementation roughly applies the same segment reordering made by the loader. This reordering moves smalls segments if they fit in alignment paddings created by other TLS segments. In case of GNU variant, for instance, the process module (id 1) is on the right of the segment and following libraries are concatenated to the left in order.

Static TLS segments are `mmaped` in copy on write from a single reference segment. As a consequence, data are allocated only when the segment is modified. Copy on write is crucial as the multiple levels of TLS require the same TLS segment to be mapped three times per thread. Indeed, in the ELF binary we make no difference between TLS which are all processed and offsetted like regular `__thread` TLS from the libC, all gathered in a static TLS segment strictly identical to the one of Figure 11. This behavior is required to avoid critical path divergence with non-modifiable components like the loader. Consequently, the only difference between TLS levels (at compiler output level) is the call emitted to the exTLS runtime, matching the target level. For example, if we compile a position independent code, TLS accesses will not be optimized and a process-level TLS read will be associated with a `extls_get_addr_process` call. When received in the exTLS library, the runtime will know that the offset in the *object table* is "0" for process level, the module will be passed in parameter of the call and will match the one computed by the libC loader. Using the module ID, the base offset in the static TLS segment is retrieved from the *offset table*, and the local offset (in parameter of the call) is added to this offset to retrieve the TLS for a given thread. As exTLS calls are handles the same as standard symbols emitted, the loader is able to generate matching IDs and offsets properly. This justifies why we follow the same critical path, simplifying a lot how exTLS can be integrated in existing components.

As far as dynamically loaded modules are concerned, they are not located in the static TLS segment (as not known at application startup), they are allocated separately in additional module entries (see Figure 21). The static TLS segment associated with the module is then loaded and referenced at all levels (as a given library may contain any type of exTLS). Dealing with dynamic lookup of TLS symbols, we implemented its support for calls to `dlsym` with a library handle. The library is registered upon the `dlopen` call and added to the *dynamic table*, then the symbol offset is searched in the library using ELF introspection before returning the pointer to TLS by adding it to the base of the dynamic TLS segment. An interesting case is when a function in a dynamically loaded library refers to a TLS entry (through a function call being in a position independent code). In this case, the module id will overflow the *offset table* (modules ids are generated in order by the loader) and therefore the runtime simply substract the number of static modules from this value to find an offset in the *dynamic table*. It can then directly apply the module offset inside the pointed TLS segment.

Looking at Figure 21, it is possible to see how inheritance was implemented, simply by cloning and duplicating the levels associated with the target inheritance level in the *object table*, this call replacing current thread's TLS context. Moreover, note that it supposes that, by default, exTLS are preserved between threads, the creation of individual contexts now being explicit except for the thread level which is always defined at the execution-stream level. Eventually, in addition to the low-level TLS handling, our exTLS runtime had to provide some additional features to correctly transpose processes to task-containers. In particular, our library provides its own implementation of the atexit function call, invoking it for tasks instead of the main OS Process. Similarly, C++ destructor for `thread_local` objects had to be redirected to our runtime. Such support appeared to be compulsory to successfully privatize common libraries such as HDF5 (atexit) and TBB (global object destructors).

We have presented the runtime implementation of our TLS model, introducing its components and data-structures. Next Section is going to focus on the work done in the linker in order to preserve the support for optimized TLS.

4.4 Linker-Level Extensions

The linker plays an important role in the optimization of TLS during the relocation process. It replaces invocations of the TLS runtime through function calls with register based offsetting as previously discussed in Section 4.1. Normal TLSs are relying on `fs` as a segment register to directly point to TLS addresses, as for example in the LE optimization level. In our case, we implemented TLS optimizations by relying on the `gs` register, which is free on `x86_64` architecture, this allows our TLS to be used conjointly with libC TLS, relying on the `fs` register. The implementation of these optimization levels has been greatly simplified by our choice of preserving the layout of the static TLS segment. We did not need to rewrite all the GOT inferences generated for "normal" TLSs as the offset calculated for the original TLS model is still valid in the exTLS model. We modified assembler generation to include the notion of level, for example, if the target is the task level, the `gs` register is offsetted to the correct cell in the *object table*. Then the pointer to the matching static TLS segment is dereferenced before being offsetted (either from the GOT or directly inline in the code) to retrieve the TLS pointer. This support for optimized TLS requires that static TLS modules be allocated prior to the execution as direct accesses are possible, this is done in copy and write in order not to waste memory particularly as only a subset of the TLS contained in the static TLS segment may be used at a given level. These modifications allowed us to support all the TLS optimization levels present in the GNU-linker, enabling efficient exTLS support.

Another aspect that we had to modify in the linker is dynamic symbol export in the case of runtime resolved functions: remote initializers `___ex_TLS_w_X`. Indeed, in the main binary (static program) these functions may not be exported to the dynamic symbol table and therefore not resolvable through `dlsym`. We modified the linker to export functions matching this name class in the dynamic symbol table, including those of the main binary, allowing us to rely on `dlsym` to resolve these symbols with a correct handling of library ordering – point which would have been much more complex to implement otherwise.

This concludes our discussion of linker-level modifications which remained quite reduced thanks to our approach of preserving the layout of the static TLS segment. Now that

we covered all the components of our exTLS implementation, allowing the task-container abstraction, we are going to present a sample use case using the MPI runtime for which privatized TLS were developed, MPC.

5. USING TLS IN MPC

The library and the facilities we develop in this paper find their origin in the extended TLS of the MPC runtime[23]. As a thread-based MPI+OpenMP runtime, MPC runs MPI Processes inside threads (to be more specific, user-level threads), rather than UNIX processes. Running inside user-level threads has several advantages[23, 16], among which are:

- Abstracting the underlying kernel, avoiding system calls;
- Avoiding busy waiting, allowing other threads to be scheduled instead;
- Allowing over-subscription of hardware resources, for example, by running hundreds of MPI tasks on a single core;
- Limiting the number of processes, shared libraries to load, communication buffers to allocate and overall launch time (one process per node instead of per core).

However, in order to benefit from these improvements, the code has to be "transposed" to threads, in particular, requiring global variables to be handled. Indeed, such variable are by default shared between all threads, and can result in execution behavior which differs from what is expected by an MPI code running inside UNIX processes. Because most MPI applications are developed with Process-based MPI in mind, it was clear that requiring users to remove global variables was not practical. Instead, techniques based on automatic compiler privatization are necessary in order to port existing codes. This led to the development of the extended TLS support for purposes of running unmodified MPI codes inside user-level threads.

In addition, it appeared that variables should not only be privatized at thread level, but that the TLS library would have to handle a hierarchy of TLS context. A simple example of hierarchical context is when an OpenMP parallel region is executed in the context of an MPI Process itself running inside a thread. The developer expects a `rank` variable with the MPI rank to be the same between all threads of the same team. A similar issue appears with the OpenMP thread-private clause which transposes to a copy per user-level thread (and not per MPI process). This support[4] has been developed inside MPC and validated using NAS-MZ benchmarks. It is now provided inside the exTLS library in order to allow hybrid MPI+X programming. Note that the exTLS library does provide the `__openmp` keyword which is equivalent to the thread-private level for convenience.

Hierarchical contexts are also of interest in relation to memory consumption. Indeed, there are several codes which depend on large constant tables, for example, to define material state or molecular proprieties. A process-based MPI code would require the use of a shared-memory window to store such data. This is not straightforward, particularly as it forbids the direct use of global variables requiring dedicated code. With MPC defined on top of extended TLS, the Hierarchical Local Storage (HLS) [26] language extensions allow variables to be topologically located. Using `pragmas` a developer may portably move a variable to a shared memory location. For example, a single node-level variable could be used instead of n copies, with n the number of cores per node. HLS also provides synchronization primitives to collectively update shared variables. On the contrary, a developer may duplicate a variable at cache level in oder to limit the false-sharing ratio. HLS allows this as well. HLS was validated with several benchmarks and representative codes[26]. Improvements with respect to cache efficiency (replication) and to memory usage (factorization) were demonstrated. The exTLS library does provide these functionalities using the pragma interface.

In this section, we have shown three direct applications of the exTLS library, validated in the context of the MPC thread-based runtime: (1) privatization, (2) OpenMP support in user-level thread context [4], and (3) topological storage[26]. These features are now contained in the exTLS library shipped with MPC[5], bringing previous developments and the contributions of this paper in a documented manner to the community.

6. CONCLUSION

In this paper we introduced the notion of *task containers*, defining a new TLS level between `__thread` and global variables. We first illustrated the advantage of such an approach, with In-Situ, IO related examples and shared-runtimes. This method allows processing to be combined simply by linking or preloading shared-libraries. Then, we introduced context inheritance which aims at building containers inside a shared-memory process, defining the associated interface and the context swapping principle, providing an efficient abstraction for shared-memory collocation. Next, we presented in detail our extended TLS implementation in contrast with the TLS support currently present in the GNU-libC. We described our extensions to the C language and how these keywords are handled. Then we presented the exTLS library explaining its operation and structure. Eventually, after explaining how we preserved linker-level optimization we briefly illustrated the integration of the exTLS library in the MPC thread-based MPI implementation, illustrating task-containers in an MPI context.

By outlining this intermediate task-container level at language and TLS level, our purpose was to provide the missing component for runtimes willing to emulate or transpose OS processes inside threads. Our initial example has, of course, been the MPC thread-based MPI runtime, but as we illustrated at the beginning of this paper, we are convinced that by changing the way MPI Process boundaries are defined, new interesting uses of MPI + X could be defined. For example, a GPU runtime could be shared between multiple tasks, a staging IO layer could be addressed directly, In-Situ analytics could borrow MPI contexts, an OpenMP runtime could be shared between multiple libraries (outside of the MPI container), all this using the inheritance principle and the added possibility of gathering threads in a task-container. This has the effect of simplifying model mixing which was previously viewed in a *stacked* manner due to the compulsory MPI container (launch layer). In our model, this is no longer true as processing could be defined alongside the computation. The exTLS library will be distributed in open-source with its associated patched GCC compiler, linker and privatization plugin from the MPC website[5] as

part of the MPC runtime. It aims at providing its abstractions to a wide variety of runtimes and applications. Our purpose is to ease the expression of collocated processing in shared-memory context by introducing a runtime agnostic interface. Interface which may find a direct application when considering the transition of existing codes to new MPI contexts such as endpoints or Sessions.

7. FUTURE WORK

Considering Figure 2 and its multiple `main` functions, we want to investigate the possibility of defining multi-main binaries. Thanks to the exTLS model, multiple libraries containing individual mains could be used to build a constellation of services collaborating inside a shared-memory context in task-containers. Moreover, by introducing a resource negotiation phase during sub-main launch, libraries could dynamically negotiate resources in the context of a many-core node. We see such extension as a promising way of expressing In-Situ analytics in a runtime agnostic fashion.

8. ADDITIONAL AUTHORS

Allen D. Malony (ParaTools Inc., `malony@paratools.com`)

9. REFERENCES

[1] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, March 2007.

[2] J.-B. Besnard, A. Malony, S. Shende, M. Pérache, P. Carribault, and J. Jaeger. An mpi halo-cell implementation for zero-copy abstraction. In *Proceedings of the 22Nd European MPI Users' Group Meeting*, EuroMPI '15, pages 3:1–3:9, New York, NY, USA, 2015. ACM.

[3] J. B. Besnard, M. Pérache, and W. Jalby. Event streaming for online performance measurements reduction. In *2013 42nd International Conference on Parallel Processing*, pages 985–994, Oct 2013.

[4] P. Carribault, M. Pérache, and H. Jourdren. *Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC*, pages 1–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[5] CEA/ParaTools. MPC Website. http://mpc.hpcframework.paratools.com/, 2016.

[6] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA, 2010. ACM.

[7] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[8] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling mpi interoperability through flexible communication endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 13–18, New York, NY, USA, 2013. ACM.

[9] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, W. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. van der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[10] M. Dorier, M. Dreher, T. Peterka, J. M. Wozniak, G. Antoniu, and B. Raffin. Lessons learned from building in situ coupling frameworks. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV2015, pages 19–24, New York, NY, USA, 2015. ACM.

[11] U. Drepper. Elf handling for thread-local storage. Technical report, Technical report, Red Hat, Inc. http://people.redhat.com/drepper/tls.pdf, 2013.

[12] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine. Hybrid mpi: Efficient message passing for multi-core systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 18:1–18:11, New York, NY, USA, 2013. ACM.

[13] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, and C.-C. Ma. Ownership passing: Efficient distributed memory programming on multi-core systems. *SIGPLAN Not.*, 48(8):177–186, Feb. 2013.

[14] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller. Mpi runtime error detection with must: Advances in deadlock detection. *Sci. Program.*, 21(3-4):109–121, July 2013.

[15] T. Hoefler and A. Lumsdaine. Message progression in parallel computing - to thread or not to thread? In *2008 IEEE International Conference on Cluster Computing*, pages 213–222, Sept 2008.

[16] C. Huang, O. Lawlor, and L. V. Kalé. *Adaptive MPI*, pages 306–322. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[17] Intel. User and Reference Guide for the Intel C++ Compiler 14.0. https://software.intel.com/en-us/node/513001, 2014.

[18] I. ISO. Iec 9899: 2011 information technology-programming languages-c. *International Organization for Standardization, Geneva, Switzerland*, 27:59, 2011.

[19] J. Jeffers and J. Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[20] R. Latham, W. Gropp, R. Ross, and R. Thakur. *Extending the MPI-2 Generalized Request Interface*, pages 223–232. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[21] N. Liu, J. Cope, P. H. Carns, C. D. Carothers, R. B. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Proceedings of MSST/SNAPI 2012*, Pacific Grove, CA, 04/2012 2012.

[22] S. Negara, G. Zheng, K.-C. Pan, N. Negara, R. E. Johnson, L. V. Kalé, and P. M. Ricker. *Automatic MPI to AMPI Program Transformation Using Photran*, pages 531–539. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[23] M. Pérache, H. Jourdren, and R. Namyst. *MPC: A Unified Parallel Runtime for Clusters of NUMA Machines*, pages 78–88. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[24] H. Sankaranarayanan and P. A. Kulkarni. Source-to-source refactoring and elimination of global variables in c programs. *Journal of Software Engineering and Applications*, 2013.

[25] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12(3):66–73, 2010.

[26] M. Tchiboukdjian, P. Carribault, and M. Pérache. Hierarchical local storage: Exploiting flexible user-data sharing between mpi tasks. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 366–377, May 2012.

[27] P. Tu and D. Padua. Compiler optimizations for scalable parallel systems. In S. Pande and D. P. Agrawal, editors, *Compiler optimizations for scalable parallel systems*, chapter Automatic Array Privatization, pages 247–281. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[28] S. Wienke, P. Springer, C. Terboven, and D. an Mey. *OpenACC — First Experiences with Real-World Applications*, pages 859–870. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[29] G. Zheng, S. Negara, C. L. Mendes, L. V. Kale, and E. R. Rodrigues. Automatic handling of global variables for multi-threaded mpi programs. In *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11, pages 220–227, Washington, DC, USA, 2011. IEEE Computer Society.