2017 IEEE International Parallel and Distributed Processing Symposium

Accommodating Thread-Level Heterogeneity in Coupled Parallel Applications

Samuel K. Gutiérrez*†, Kei Davis*, Dorian C. Arnold†, Randal S. Baker*, Robert W. Robey*
Patrick McCormick*, Daniel Holladay*, Jon A. Dahl*, R. Joe Zerr*, Florian Weik*, and Christoph Junghans*
*Los Alamos National Laboratory, Los Alamos, New Mexico 87545
†Department of Computer Science, University of New Mexico, Albuquerque, New Mexico 87131
Email: samuel@lanl.gov

***Abstract*—Hybrid parallel program models that combine message passing and multithreading (MP+MT) are becoming more popular, extending the basic *message passing* (MP) model that uses single-threaded processes for both inter- and intra-node parallelism. A consequence is that coupled parallel applications increasingly comprise MP libraries together with MP+MT libraries with differing preferred degrees of threading, resulting in *thread-level heterogeneity*. Retroactively matching threading levels between independently developed and maintained libraries is difficult; the challenge is exacerbated because contemporary parallel job launchers provide only static resource binding policies over entire application executions. A standard approach for accommodating thread-level heterogeneity is to under-subscribe compute resources such that the library with the highest degree of threading per process has one processing element per thread. This results in libraries with fewer threads per process utilizing only a fraction of the available compute resources.**

We present and evaluate a novel approach for accommodating thread-level heterogeneity. Our approach enables full utilization of all available compute resources throughout an application's execution by providing programmable facilities to dynamically reconfigure runtime environments for compute phases with differing threading factors and memory affinities. We show that our approach can improve overall application performance by up to 5.8x in real-world production codes. Furthermore, the practicality and utility of our approach has been demonstrated by continuous production use for over one year, and by more recent incorporation into a number of production codes.

***Keywords*-MPI, MPI+X, Pthreads, OpenMP.**

I. Introduction

Parallel and distributed applications such as multi-physics applications play crucial roles in science and engineering. Because of their interdisciplinary nature these applications are often *coupled*, that is, built via the integration (or coupling) of independently developed and tuned software libraries linked into a single application. In such coupled applications, a poorly performing library can lead to overall poor application performance and increased time-to-solution. It is critical that each library is executed in a manner consistent with its design and tuning for a particular system architecture and workload. Generally, each library (*input/compute phase pair*) has its own optimal *runtime configuration*, for example, number of processes or threads. In coupled applications, effective configuration parameters are determined (most often heuristically, manually, and offline) for all performance-critical computational phases. *Configuration conflicts* arise when an optimal configuration for one phase is suboptimal for another. There are a variety of approaches for resolving configuration conflicts. At one extreme lie applications written to parallel and distributed programming systems such as Legion [1] and Charm++ [2], which by design resolve such conflicts at runtime. At the other extreme lie *MP+MT* applications that use message passing (MP) for inter- and intra-node parallelism and multithreading (MT) for additional intra-node parallelism, where the common approach is to allocate resources to satisfy the most demanding compute phase. The library with the highest degree of threading per process has one processing element per thread, and libraries with fewer threads per process run *under-subscribed*, using only a fraction of the available compute resources when running.

In this work we study coupled MP+MT applications with dynamic, phased configuration conflicts. Focusing on applications based on the Message Passing Interface (MPI)[1] [3], we address the practical challenges of *thread-level heterogeneity*, where a coupled application comprises MPI libraries that require different degrees of thread-level parallelism. We present a general methodology and corresponding implementation for dynamically (at runtime) accommodating coupled application configuration conflicts in a way that 1) is composable, 2) is hardware topology aware, 3) is MPI implementation agnostic,[2] 4) works with a variety of unmodified Pthread-based parallel programming systems, 5) increases overall system resource utilization, 6) reintroduces lost parallelism, and 7) is straightforward to incorporate into existing applications. To the best of our knowledge this is the first work to satisfy all of these criteria. Finally, we evaluate our methodology by applying it to three production-quality simulation codes that employ a variety of parallelization strategies. Our results show that significant performance improvements can be achieved when used in environments positioned to make effective use of the additional levels of parallelism our approach enables.

[1]MPI has been the predominant scientific parallel programming system for the last two decades.

[2]So long as the underlying representation of an MPI process is a system process. This is true for most MPI implementations.

1530-2075 2017

DOI 10.1109/IPDPS.2017.13

II. Coupled Applications and their Challenges

As previously described, parallel applications are often built by coupling independently developed and tuned software libraries. For example, coupled physics applications are often implemented in a fashion where each physics library, in turn, updates common application state data. Such scientific libraries tend to have their own preferred data discretization scheme, for example, unstructured meshes, regular meshes, or particles, so they manage their own distributed state and parallelization strategies with little or no coordination across library boundaries. More generally, libraries interact by exchanging data through application programming interfaces (APIs) that remap data from one library domain to another, for example, from a field defined on a computational mesh to a system of linear equations, or from one mesh to another as illustrated in Figure 1. Quite often, such data structure remappings suggest complementary remappings of tasks (processes/threads) to hardware. Inter-library interactions can take place many times during the lifespan of an application. Furthermore, at a given program point these interactions may change during the course of a simulation to accommodate new requirements, for example, particular physics appropriate for a particular spatial scale and resolution.

A. Coupled Application Parallelism

Parallel scientific application executions exploit data parallelism, where many instances of the same computation execute in parallel on different data and on different computational resources. In the pure MP model, message passing is used for both inter- and intra-node parallelism (other than SIMD vectorization). For MPI applications this is called *MPI-everywhere*. In this model, computational resources are usually *fully subscribed*, that is, the program's set of single-threaded processes is in one-to-one correspondence with processing elements (PEs), i.e., cores or hardware threads, and parallelism is realized via either *single program, multiple data* (SPMD) or *multiple program, multiple data* (MPMD) schemes.

Alternatively, a scientific application can employ a hybrid model using MP plus multithreading (MP+MT) for inter- and intra-node parallelism, respectively. For MPI applications, MP+MT is an instance of the more general *MPI+X* model in which applications employ additional on-node parallelization strategies. This approach is increasingly popular as core (or hardware thread) counts increase in shared-memory nodes, and because of the flexibility and performance potential of a hierarchical approach [4], [5], [6].

While MPI+X is gaining popularity it is not yet ubiquitous. Restructuring large, mature code bases to effectively exploit new parallel programming systems is challenging and generally requires a significant amount of effort that is often unjustifiable because of cost or priority. Furthermore, it is not uncommon that an MPI-everywhere version of a scientific code performs as well or better than its MPI+X instantiation [7], [8], which discourages speculative hybridizing of existing codes. Finally, while an MPI+X library may be written such that its runtime configuration is settable within some range at startup, the particular runtime parameters that give best performance may depend on static or dynamic variables such as input and problem scaling factors. For all of these reasons, coupled scientific codes will for the foreseeable future continue to be built from libraries that use a mix of non-uniform runtime configurations. A runtime configuration may include the total number of processes to use for SPMD or MPMD parallelism, a process threading degree for shared-memory multithreading, and a mapping of tasks (processes and threads) to compute resources, e.g., PEs and memories.

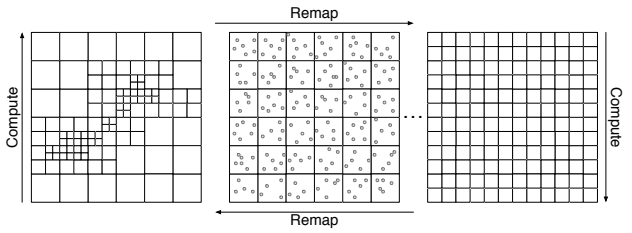


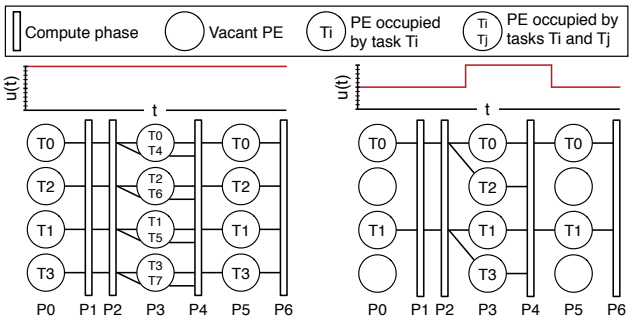
Figure 1: Notional illustration of computational phases interleaved with data structure remapping phases across domains.

B. Coupled Applications with Conflicting Configurations

For decades coupled applications had relatively uniform library configuration requirements because they were built from single-threaded libraries, so static configurations set at application launch were sufficient. Today, however, configuration conflicts are common in coupled applications because they comprise independently developed and maintained scientific libraries that have been written or ported to hybrid MP+MT programming models.

1) Static Configurations: In today's static computing environments, dynamically accommodating inter-library configuration conflicts is difficult. While it is well-understood that binding tasks to hardware resources can improve the performance of an MPI application [9], [10], parallel application launchers such as orterun, srun, aprun, and Hydra only allow static allocations and static binding capabilities: launch-time configurations persist for the entire parallel application's execution. Most single-threaded applications are launched by binding a single PE dedicatedly to each process. This mitigates the ill effects of task migration in multiprocessor architectures, for example, cache invalidation that occurs when a thread moves from one PE to another.

With a static configuration for coupled MP+MT applications with conflicting configurations, the two basic configuration options are *over-subscription* and *under-subscription*. In over-subscribed configurations, all allocated resources are



(a) Time evolution of a static over-subscribed MPI+X configuration.

(b) Under-subscribed MPI+X with typical *wide* binding policy.

Figure 2: Illustration of compute resource utilization $u(t)$ by tasks (processes and threads) over time for two MPI+X configurations.

always in use, i.e., the number of PEs equals the number of threads in the computational phase with the lowest degree of threading per process. In phases that require higher numbers of threads, resources are over-subscribed with multiple threads per PE. Figure 2a illustrates the evolution of an *over-subscribed* MPI+X configuration where hardware utilization, $u(t)$, remains constant at 100%. In this example MPI-only phases fully-subscribe hardware resources (phases P0-P2, P5-P6), while multi-threaded regions over-subscribe them (phases P3-P4). In practice over-subscription is generally avoided because the increased resource contention in threaded regions tends to negatively affect overall application performance and scalability [11].

The standard approach for accommodating thread-level heterogeneity in coupled MPI applications is to statically (at launch time) under-subscribe compute resources such that the computational phase with the highest degree of threading per MPI process has one PE per software thread. As a consequence, phases with fewer threads per process use only a fraction of the available compute resources, thus leading to poor system resource utilization. Figure 2b illustrates the evolution of compute hardware resource utilization over time for a typical MPI+X configuration. Over time, hardware utilization fluctuates between 50% and 100% as the application moves in and out of regions with differing degrees of multithreading. For phase-homogeneous MPI+X-only applications—ones with relatively static workload characteristics—under-subscription is reasonable when the cost of lower coarse-grained data parallelism is outweighed by the benefits of introducing multi-threading.

2) Lost Parallelism via Resource Under-Subscription: Given an application that strong-scales perfectly (the theoretical upper bound), we can calculate the theoretical slowdown of static under-subscription approaches using Amdahl's law,

$$S = \frac{1}{\sum_{i=1}^{m} \frac{p_i}{s_i}} \tag{1}$$

where n is the total number of available processors; m is the total number of phases; t_i is the optimal threading degree

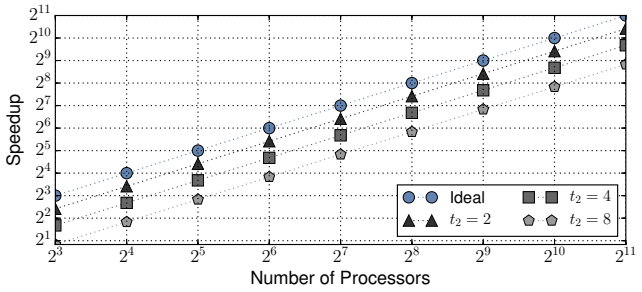


Figure 3: Log-log plot of modeled speedups.

for a phase i; $t_{\max} = \max(t_1, \ldots, t_m)$; $u_i = t_i / t_{\max}$ is a phase's processor under-subscription factor; and $s_i = n \cdot u_i$ is the speedup factor for a given phase. Consider two serial phases L1 and L2 whose percentages of execution time are equal: $p_1 = 0.5$ and $p_2 = 0.5$. Assuming that L1 runs optimally with an MPI-everywhere parallelization strategy and L2 optimally with an MPI+X strategy, Figure 3 plots the theoretical speedups of three under-subscribed runtime configurations where L1's threading degree is fixed at $t_1 = 1$ and L2's varies. We compare those to an idealized configuration (Ideal) where each phase of the parallel computation is exposed to all available processors. This simple model illustrates the potential losses in performance that can result from today's static under-subscription approach.

In summary, coupled scientific applications based on the MP+MT model can comprise libraries with conflicting configuration requirements. For such applications, today's static computational environments necessitate suboptimal over-subscribed or under-subscribed configurations. Therefore there is a need for techniques that address dynamic, conflicting configurations in coupled MP+MT applications.

III. Dynamic MPI+X with QUO

Next we present a general, composable runtime approach for programmatically accommodating library configuration conflicts that arise in dynamic, coupled, thread-based MPI+X applications. Our design is influenced by requirements for generality, composability, efficiency, and pragmatism in the face of production realities, that is, easily fitting into large, established code bases that may still be under active development.

QUO (as in *"status quo"*) is both a model and a corresponding implementation that facilitates the dynamically varying requirements of computational phases in coupled MP+MT applications. Specifically, QUO allows an application to dynamically query and reconfigure its process bindings. While the model is general, the current implementation focuses on Pthread-based MPI+X applications [12]. Fundamentally, QUO uses hwloc [9] and MPI, interfacing with those libraries and the application as shown in Figure 4. The hwloc library is used to gather system hardware topology information and to control process binding policy changes during the target application's lifetime. MPI is used

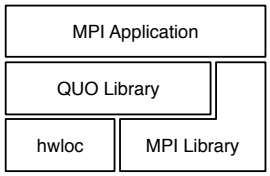


Figure 4: QUO architecture diagram.

primarily for exchanging setup information during *QUO context* (QC) setup, which is discussed in a later section.

The portable, production-quality, open-source runtime library is written in C, but also provides C++ and Fortran language bindings. The QUO API operates on QC pointers. This design allows for the creation and manipulation of multiple QCs within a single application that are either encapsulated within a library or passed from one library to another—a key for composability. The remainder of this section presents the principle concepts and mechanisms that underlie our design and approach.

A. QUO Contexts

QUO contexts, which encapsulate state data gathered and manipulated by QUO, are created via a collective call to `QUO_create()` in which all members of the initializing communicator must participate. QCs may be created at any time after the underlying MPI library has been initialized and remain valid until freed via `QUO_free()`, which must occur before the MPI library has been *finalized*. QUO can maintain multiple independent, coexisting QCs within a single application.

B. Hardware/Software Environment Queries

Contemporary HPC node architectures are complex and diverse, demanding careful consideration of their resource (PE and memory) configurations. To effectively guide the dynamic (runtime) mapping of application-specific software (logical) affinities to hardware resources, one must be able to obtain both the underlying platform's resource information and the application's current usage of those resources. In this regard, QUO's approach is straightforward: its API provides thin convenience wrappers around commonly-used hwloc hardware query routines for hardware information. Relevant hardware information includes memory localities relative to PEs in non-uniform memory access (NUMA) architectures and hierarchical hardware relationships (e.g., determining how many cores are contained in a particular socket).

Process affinity state queries provide another mechanism to influence runtime software-to-hardware mapping decisions based on the hardware affinities of cooperating processes within a compute node. For example, on a per-node basis, one can query for the set of MPI processes with affinity to a particular hardware resource. In Linux, a thread's CPU affinity mask determines the set of CPUs on which it is eligible to run. For these queries, QUO uses a

```
1  QUO_create(&ctx, MPI_COMM_WORLD) // Create a context
2  // Query runtime software/hardware environment to
3  // influence runtime configuration selection algorithm
4  ...
5  tres = NUMA_NODE // Set target resource to NUMA
6  // Let QUO determine a set of node-local MPI processes
7  // (optimized for maintaining data locality) that
8  // satisfy the distribution criterion that no more
9  // than max_pe processes be assigned to each NUMA
10 // domain in the host
11 QUO_auto_distrib(ctx, tres, max_pe, &in_dset)
12 // If in_dset is true, then the calling process is a
13 // member of the distribution set
14 if (in_dset)
15     // Expand affinity to cover resources with
16     // affinity to caller's closest NUMA domain
17     QUO_bind_push(ctx, tres)
18     // Perform coupled threaded computation with newly
19     // enacted process hardware affinity policy
20     A_coupled_threaded_library_call(in_args, &result)
21     // Revert to prior process binding policy before
22     // entering node-local QUO barrier
23     QUO_bind_pop(ctx)
24 // Quiesce set of active MPI processes not in
25 // distribution set by yielding their use of compute
26 // resources, while those who are spawn threads onto
27 // those resources
28 QUO_barrier(ctx)
29 // Barrier complete, all MPI processes participate in
30 // result dissemination (via message passing) to relay
31 // result to all cooperating processes in calculation
32 ...
33 QUO_free(ctx) // Relinquish context resources
```

Example 1: A *caller-driven* policy example using hardware queries and application characteristics to guide a dynamic affinity policy using QUO (C API pseudocode).

combination of hwloc and MPI services. For a given QC, QUO uses MPI to share a cached mapping of MPI processes to process IDs, and hwloc is used to query the affinities of the relevant processes. Note that to effectively map tasks to PEs, both intra-process (first party) and inter-process (third party) affinity state queries are necessary.

C. Programmable Dynamic Process Affinities

QUO allows arbitrary process binding policies to be enacted and reverted during the execution of an MPI+X application. Ideally, binding instantiations and reversions will coincide with the entries and exits of the application's different computational phases. Accordingly, QUO exposes a simple, stack-based semantics through `QUO_bind_push()` and `QUO_bind_pop()`. For example, a new process binding policy can be instantiated before entering a threaded computational phase via `QUO_bind_push()` and then reverted at the end of that phase via a `QUO_bind_pop()`. These semantics allow a user to conveniently and dynamically stack an arbitrary number of binding policies that map to the potentially stacked composition of coupled components in a QUO-enabled MPI+X application (Listing I and Figure 5).

QUO offers two variants of `QUO_bind_push()`. The first pushes a hardware affinity policy specifically outlined by the caller. This variant unconditionally, without regard to the caller's current hardware affinities, changes the calling process's affinity mask to encompass the PEs dominated by the provided policy. QUO also provides a more sophisticated

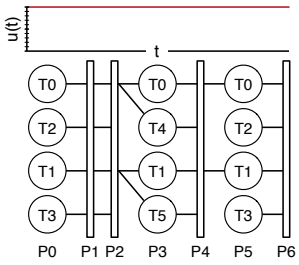


Figure 5: Illustration of compute resource utilization by tasks over time $u(t)$ for a QUO-enabled MPI+X configuration.

version of this call that first queries the caller's current hardware affinities to choose the *closest* target resource that dominates the caller's current hardware affinities in hwloc's hardware object tree.[3] If, for example, the caller currently has an affinity to a core in socket 3, then a call to the latter variant with a target resource of `SOCKET` will automatically expand the caller's affinity mask to encompass all PEs within socket 3. The rationale for this functionality is to maintain data locality (that is, memory affinity) while moving in and out of process binding policies, in this case keeping data resident within one NUMA region across library calls.

Intra- and inter-process affinity state queries are used to guide dynamic binding policy choices and are often used in concert with `QUO_bind_push()`. For added convenience, QUO offers an automatic task distribution capability via `QUO_auto_distrib()`. This routine automates the two-step query and bind process at the cost of generality. Specifically, this routine allows callers the ability to evenly distribute tasks across a specified resource with minimal effort. For example, one can specify that a maximum of two tasks be assigned to each socket on the target compute resource, and this routine will do so by choosing at most two tasks that are enclosed within (that is, have an affinity to) each respective socket. When there exists a subset of cooperating processes not *bound* to a particular hardware resource, QUO first favors *bound* processes, adding *unbound* processes only if the distribution criteria were not satisfied with *bound* processes. This maintains data locality when moving in and out of process binding policies, easing programmer burden. With these primitives, applications can dynamically create policies tailored specifically to their current needs based on the underlying hardware characteristics and the current process binding policies of other participating processes within a compute node.

D. Node-Level Process Quiescence

To make our approach maximally effective there must be a portable and efficient mechanism for *quiescing* sets of MPI processes to yield their use of compute resources to *make room* for more threads of execution, as illustrated in Listing I and Figure 5. A naive approach might use MPI-provided facilities such as an `MPI_Barrier()` across a sub-communicator containing only processes that may communicate over shared memory, for example, a sub-communicator created by calling `MPI_Comm_split_type()` with `MPI_COMM_TYPE_SHARED` as its split type. While this approach certainly works (as demonstrated in an early implementation of QUO), it introduces prohibitively high overheads and is therefore unusable in practice (an analysis of process-quiescence-induced application overhead is presented in Section IV-B). Instead, we employ an efficient, portable approach for compute-node-level process quiescence via `QUO_barrier()`. Its underlying machinery is straightforward:

1) At `QUO_create()` a shared-memory segment is created by one MPI process and then attached to by all processes P that are a) members of the initializing communicator and b) capable of communicating over shared memory.
2) A `pthread_barrier_t` is embedded into the shared-memory segment with an attribute that allows all processes with access to the shared-memory segment to operate on it. Finally, its count parameter is set to the number of MPI processes that must

P0	Four single-threaded processes $P = \{T_0, T_2, T_1, T_3\}$ are launched onto cores $R = \{C_0, C_1, C_2, C_3\}$, where each process $T_{0,\ldots n-1} \in P$ has affinity to the core on which it was launched: T_0/C_0, T_2/C_1, T_1/C_2, T_3/C_3. Process state data $S = \{M_0, M_2, M_1, M_3\}$ is initialized for each process in P.
P1	Processes in P execute in parallel during first compute phase, fully subscribing the compute resources in R.
P2	Processes in P map data from their domain X (resident in S) to the callee's domain Y, $\mathbb{M} : X_m \rightarrow Y_n$, where $m = \lvert P\rvert$ and $n = \lvert L\rvert$. Then two processes in P, namely $Q = \{T_2, T_3\}$, are quiesced while the remaining processes $L = P - Q$ *push* a new binding policy such that their hardware affinities expand to cover two cores each: $T_0/C_0 \lvert\lvert C_1$, $T_1/C_2 \lvert\lvert C_3$. State in M_0 is now shared between T_0, T_4, while M_1 in a similar fashion is shared between T_1 and T_5.
P3	Two new threads $\hat{P} = \{T_4, T_5\}$ are spawned by their respective parents in L onto cores C_1, C_3, namely cores once occupied by MPI processes in Q.
P4	Processes and threads residing in $L \cup \hat{P}$ execute in parallel during this compute phase, fully subscribing the compute resources in R. The threaded compute phase completes and the spawned threads in $\hat{P}$ die or are suspended by the threading library's runtime. Processes in L revert to their previous binding policies by *popping* them off their respective affinity stacks.
P5	Processes in Q resume execution on the computational resources they relinquished in P2.
P6	Processes in P map data from domain Y (resident in $\hat{S} = \{M_0, M_1\}$) back to the caller's domain X (residing over state in S), $\mathbb{M} : Y_n \rightarrow X_m$, where $n = \lvert L\rvert$ and $m = \lvert P\rvert$. That is, results are disseminated via explicit message passing from n processes in L to m processes in P.

Listing I: Explanation of QUO-enabled MPI+X phases in Figure 5.

[3]hwloc represents a hardware topology as a dominance tree of logical hardware objects. For example, the root is a machine object that dominates all other system resource objects.

call `pthread_barrier_wait()` to complete the barrier, i.e., the number of processes in P.

E. Data Dependencies

Before the number of active MPI processes can be safely increased or decreased, data must be exchanged among node-local processes to satisfy all inter-process data dependencies. Typically, this occurs via node-local gathers and scatters before and after QUO-enabled regions as described in Listing I (P2 and P6). As is typical for message passing models, inter-process data dependences are managed explicitly and programmatically. Once dependencies are satisfied, QUO can enact arbitrary task reconfigurations.

F. Policy Management

Policies that influence how logical (software) affinities are mapped to hardware resources may be managed with QUO in a variety of ways. In a *caller-driven* approach, as shown in Example 1, the caller modifies the callee's runtime environment and assumes responsibility for resource selection (the computational resources to be used by a particular computational phase), MPI process selection (the set of MPI processes that will use the selected resources), and process affinity selection (*pushing* and *popping* binding policies as the target library's computational phases are entered and exited, respectively). A caller-driven approach is appropriate when using off-the-shelf threaded libraries that are difficult or impossible to modify at the source code level. This requires the caller to be cognizant of the inner workings of the target library to make informed policy decisions. In contrast, *callee-driven* policies are encapsulated within called libraries such that the caller may be oblivious to policy decisions made by the libraries it uses. Because these policies are directly embedded in the target library and are under developer control they can be precisely tailored to the library's implementation and runtime requirements.

IV. QUO Performance and Effectiveness

Our performance evaluation is designed to show performance and scaling characteristics for both QUO micro-operations and for full applications. For the former, we study the costs of the key QUO operations. For the latter, we integrate QUO into three different production-quality parallel scientific applications using a variety of parallelization strategies. With these we measure and analyze QUO's costs and benefits, and how these vary with scale. Integrating QUO into three diverse codes demonstrates the generality of the QUO approach.

A. Micro-Benchmark Results: Cost of QUO Operations

We quantify the individual overhead costs for a representative set of QUO operations. For each operation we measure the time required to complete that operation 100 times in a tight loop—at each scale, processes co-located on the same compute node simultaneously execute this loop. Micro-benchmark results were collected on a Cray XE6 platform built from compute nodes containing AMD 6136 processors. Further architectural details are provided in Table I.

CPU	Sockets per Node	NUMA Domains per Socket	Cores per Socket
AMD 6136	2	2	8
Intel E5–2670	2	1	8
Intel E5–2660	2	1	10

Table I: Overview of system architectures.

Figure 6 shows each operation's average execution times as a function of scale. All QUO operations, except `QUO_create()` and `QUO_free()`, are performed on a per-node basis and their overheads are a function of the number of concurrent QUO processes within a single compute node. This phenomenon is observed in job sizes ranging from one to sixteen processes since our test platform contains sixteen-core compute nodes. `QUO_create()` and `QUO_free()` overheads depend on the total number of processes in the initializing communicator because they require global (inter-node) process communication. Figure 6 shows that even beyond 16 processes (the node width) the performance of these two operations continues to grow. Even so, their costs are modest at ∼100ms at 2,048 processes across 128 nodes. Furthermore, these costs are amortized over the life of a QUO context: we expect most applications to use long-lived contexts that persist until library termination. Note that a long-lived context does not imply a single, static configuration; rather, it implies a single dynamic instance of QUO-maintained state.

B. Application Overhead from Process Quiescence

To evaluate the overhead of QUO process quiescence—a key QUO mechanism—we compare two approaches, namely `MPI_Barrier()` and `QUO_barrier()`. The benchmarking application is simple: an MPI-everywhere driver program that calls 2MESH's computationally intensive MPI+OpenMP library described in Table IV. Depending

Figure 6: Log-log plot: average execution times of a representative set of QUO operations on a Cray XE6.

on the setup, before the multithreaded computation can be executed there is either no quiescence (ideally), or quiescence using one of the two approaches. We compare wall-clock times reported by the application when using each mechanism. The single-node experiment is as follows.

1) 16 MPI processes are launched with a core binding policy that fully subscribes the compute node.
2) Four MPI processes are chosen such that each has affinity to a different NUMA domain. The processes in this set, P, will enter the threaded compute phase.
3) Before executing the threaded 2MESH phase, processes in P push a NUMA binding policy to accommodate the four OpenMP threads they will spawn, while the remaining processes are quiesced using either `MPI_Barrier()` or `QUO_barrier()`.

Table II contrasts the performance of `MPI_Barrier()` and `QUO_barrier()` with that of the ideal case in which four MPI processes (each with a NUMA binding policy) are launched across all four NUMA domains on the target architecture, thereby avoiding the need for quiescing any processes, thus mimicking what today's MPI+X codes do in practice. The results show that our `QUO_barrier()` implementation significantly outperforms `MPI_barrier()` and is close to the ideal case where quiescence is not necessary. In particular, our approach introduces approximately an 8% overhead, while the naive approach using `MPI_Barrier()` over `MPI_COMM_WORLD` introduces approximately 116% overhead.[4]

Process Quiescence Mechanism	Average Execution Time	Standard Deviation	Mechanism-Introduced Overhead
Ideal	16.46 s	0.05 s	—
QUO_barrier	17.82 s	0.32 s	8.24%
MPI_Barrier	35.49 s	0.17 s	115.63%

Table II: Quiescence-induced overhead by mechanism.

C. Application Results: Evaluating QUO's Effectiveness

Table IV details the three QUO-enabled parallel scientific applications using all of the supported language bindings (C, C++, and Fortran) and a diversity of parallelization strategies, workloads, and software environments. All application configurations represent real workloads to showcase different application communication and computation characteristics.

We evaluated QUO's effectiveness at increasing resource utilization with comparisons against a baseline (without QUO) that under-subscribes compute nodes such that the computational phase with the highest degree of threading per process ($t_{\max}$) has one PE per thread. This baseline represents the previous, long-standing mode for production runs of these applications. Table III shows the application configurations. For baseline experiments, MPI processes are launched with a static process binding policy set by either aprun (Cray-MPICH) or orterun (Open MPI). For example, 2MESH is launched with four MPI processes per node (one process per NUMA domain), each with a NUMA binding policy. In contrast, QUO-enabled experiments fully subscribe resources at startup such that each MPI process is bound to a single core (by the parallel launcher) and MPI+X configuration policies are enacted dynamically using QUO.

Identifier	MPI+X Process Binding Policy	Processes per Resource	$t_{\max}$
2MESH-W	NUMA	1/NUMA	4
RAGE-W	Machine	1/Machine	16
ESMD-W	Socket	10/Socket	2
2MESH-S	NUMA	1/NUMA	4
RAGE-S	Machine	1/Machine	16
ESMD-S	Socket	5/Socket	4

Table III: Application configurations.

QUO Performance Results: We evaluated the three QUO-enabled applications on three different platforms at scales up to 2,048 PEs (and processes).[5] Figure 7 shows all of the application performance and scaling results: 30 sets of experiments, ten different application/workload combinations, each executed at three different scales. QUO's effectiveness is determined principally by two criteria: 1) how much of an application's overall runtime is dominated by under-subscribed computational phases and 2) how well these otherwise under-subscribed computational phases strong-scale at full node utilization.

The overall average speedup across all 30 QUO-enabled workloads was $\sim$70%. Of these workloads, 26 show an overall speedup when using QUO, with more than half the cases (16) yielding speedups greater than 50%. A maximum QUO-enabled speedup of 476% is yielded by RAGE-S3 at 64 PEs and seven other workload configurations showed a speedup of greater than 100%. The reason these workloads realize huge benefits when dynamically configured using QUO is because their otherwise under-scribed computational phase (in this case the MPI-everywhere phase) strong-scales well with the given realistic input sets.

Four of the QUO-enabled workloads yield very modest speedups (less than 10%) and four other cases in fact demonstrated slowdowns (ESMD-S2–640, 2MESH-W4–128, 2MESH-W4–512, and 2MESH-W4–512). There are three main reasons for this: 1) as previously mentioned, if the under-subscribed phase does not strong-scale well, QUO's

[4]Because this experiment runs on a single compute node this setup mimics the shared-memory sub-communicator approach outlined in Section III.

[5]The seemingly strange PE counts (80, 320, 640) in the ESPResSo experiments are because they were run on a system with 40 hardware threads per node (2 hardware threads per core).

ID	Description	Environment	Architecture
2MESH	LANL-App is an application used at Los Alamos National Laboratory (LANL) comprising two libraries L0 and L1. L0 simulates one type of physics on an adaptive structured mesh and L1 simulates a different physics on a separate, structured mesh. L0 phases are MPI-everywhere; L1 phases are MPI+OpenMP.	Intel 15.0.4 Cray-MPICH 7.0.1 QUO 1.2.4	AMD 6136 32 GB of RAM Cray Gemini
RAGE	xRage+inlinIte: xRage solves the Euler equations of conservation of mass, momentum, and energy on an adaptive structured mesh. inlinIte solves for atomic populations in cases not in local thermodynamic equilibrium. xRage phases are MPI-everywhere; inlinIte phases are multi-threaded with Kokkos [13].	Intel 16.0.3 Open MPI 1.6.5 QUO 1.2.9	Intel E5–2670 64 GB of RAM Qlogic QDR IB
ESMD	MD+Analysis: ESPResSo [14] is a molecular dynamics (MD) program for coarse-grained soft matter applications. Its analysis routines typically calculate observables (functions of the current system state). MD phases are MPI-everywhere; analysis phases are MPI+OpenMP.	GCC 4.9.3 Open MPI 1.10.3 QUO 1.3-alpha	Intel E5–2660 128 GB of RAM 10 Gb Ethernet

Table IV: Target applications and their respective environments used for this study. 2MESH uses the QUO Fortran API; inlinIte is a C++ application, but uses the QUO C API; and ESMD uses the QUO C++ API.

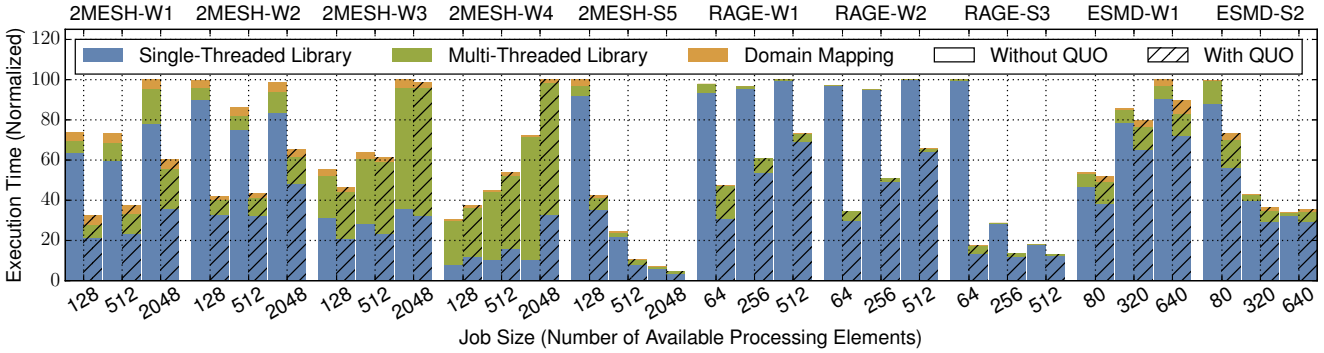


Figure 7: Application results without and with QUO. Application configurations are outlined in Table III.

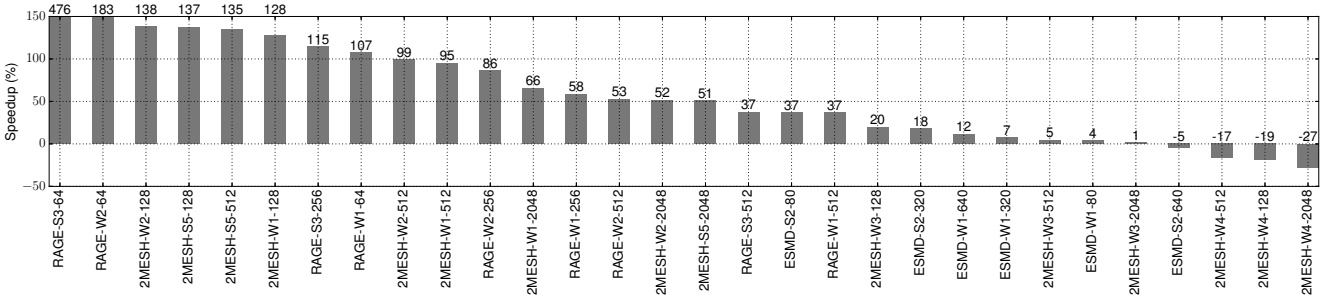


Figure 8: QUO-enabled speedups (QUO versus a standard under-subscribed baseline) for all presented workloads in Figure 7.

approach will not yield a significant performance boost; 2) QUO can increase the costs of data domain remappings; and 3) in some cases, QUO appears to add some overhead to the fully-subscribed computational phase. These phenomena can be observed in Figure 7.

V. Practical Considerations

As previously described, commonly used parallel application launchers only provide for static, whole-application binding policies, or none at all, and each has its own syntax for command-line or configuration-file specification of binding policy. Using QUO one does not need to specify binding policies via the job launcher: QUO can completely specify and manage resource bindings efficiently and dynamically. While QUO is simple, efficient, effective, and convenient, it does introduce some practical considerations and complexities:

- Increased Code Complexity: With the quiescing and later resumption of tasks, application data re-mappings across library domains may increase. Either the library developer or the library user must be prepared to deal with this added programming burden. We posit that in a well-engineered library such complexity is manageable.
- Encapsulating Code Regions: When using the QUO approach one must identify and surround computationally-intensive code regions with calls

to `QUO_bind_push()` and `QUO_bind_pop()`. Again, in a well-engineered library these modifications should be trivial—normally such code regions are well-bound by a function call or within a loop body.

- Determining Threading Levels: Though not brought by the use of our methodology, determining the minimum required threading level at `MPI_Init_thread()` can be challenging in a dynamic multi-library environment. That is, a threaded library may only execute under certain circumstances that are not necessarily evident at MPI initialization time, for example, at runtime requiring a new physics capability. Blindly initializing with the highest level of thread safety (that is, `MPI_THREAD_MULTIPLE`) is wasteful because of performance degradation brought by higher degrees of required critical section protection in an MPI library [15].

At LANL the practicality and utility of our pragmatic approach has been demonstrated by continuous production use for over one year: running at scales well in excess of 200k PEs and servicing demanding scientific workloads running on a wide variety of HPC platforms.

VI. Related Work

Hybridizing MPI applications has been studied extensively [16], [17], [18], [19]. These works suggest that choosing between MPI-everywhere and MPI+OpenMP is a non-trivial task that involves careful consideration regarding, but not limited to, algorithmic choices in the application and the characteristics of the target architecture. These works evaluate MPI+OpenMP schemes that use a static under-subscription approach, whereas we present a general methodology to dynamically accommodate a broader set of Pthread-based MPI+X schemes that consider both data and hardware localities at runtime.

Dynamic process and memory binding methodologies that consider application, data, and hardware localities have also been studied. Broquedis et al. present and evaluate hwloc by incorporating it into MPI and OpenMP runtimes to dynamically guide task affinities at runtime [9]. Unlike our work, however, their work does not present a methodology to dynamically accommodate thread-level heterogeneity in coupled Pthread-based MPI+X applications.

For HPC applications there are a variety of published approaches for efficiently resolving runtime configuration conflicts that arise in thread-heterogeneous environments. Carribault et al. present a unified runtime for both distributed- and shared-memory MPI+X codes [20]. Unlike other MPI implementations, theirs implements MPI processes as user-level threads (instead of processes), so their scheduler can efficiently accommodate both single- and multi-threaded regions during the execution of an MPI+X application. In contrast, our approach is MPI implementation agnostic and exposes an API to programmatically influence task placement and scheduling at runtime. Huang et al. present another MPI implementation that uses processor virtualization to facilitate application adaptation, including thread-level heterogeneity [21]. Their approach, however, requires the use of their runtime and modified versions of others, for example, GNU OpenMP, whereas ours works with unmodified MPI and OpenMP runtimes. Other parallel and distributed programming systems such as Legion [1] and Charm++ [2] are designed to dynamically resolve runtime configuration conflicts, but once again require that applications be rewritten to their respective paradigms.

VII. Conclusion and Future Work

We have presented a novel approach and implementation for accommodating thread-level heterogeneity in coupled MPI applications. Our approach, QUO, enables full utilization of all available compute resources throughout an application's entire execution. Its overhead can be modest and significant performance improvements can be achieved when used in environments positioned to make effective use of the additional levels of parallelism our strong-scaling approach enables. Our performance results show that for a majority of the 30 tested workloads, using QUO renders speedups greater than 50%, and the best case speedup was a resounding 476%.

QUO's interface is programmable, meaning that it can be used preferentially in cases where it will improve performance and not used otherwise in favor of a conventional static policy. Better yet, a graded approach could be used wherein only that subset of libraries that benefit from strong-scaling are strong-scaled, and to the optimal degree within the available bounds. This in turn implies that the decision to actively use QUO, and the strong-scaling factors used when it is, could be made dynamically, but we have not yet explored this possibility.

One important aspect of analysis remains, namely the precise measurement of QUO overhead. The contributions fall into three groups: QUO's effects on remappings, QUO's effects on libraries with thread scaling factors that do not change when run with QUO, and QUO's effects on libraries with thread scaling factors that are different when run under QUO. The first two analyses can be done based on data obtained from our various test cases. The third analysis requires experimentation that we have not yet performed, namely strong-scaling the target libraries in the absence of QUO. This then would allow the overhead of QUO to be separated from the effects of less than perfect strong scaling (over which QUO has no influence). This remains work for the immediate future.

VIII. Acknowledgment

Work supported by the Advanced Simulation and Computing program of the U.S. Department of Energy's NNSA. Los Alamos National Laboratory is managed and operated

by Los Alamos National Security, LLC (LANS), under contract number DE-AC52-06NA25396 for the Department of Energys National Nuclear Security Administration (NNSA).

REFERENCES

[1] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "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, 2012, p. 66.

[2] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[3] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 3.1," June 2015. [Online]. Available: http://www.mpi-forum.org/

[4] S. Habib, V. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. Insley, D. Daniel, P. Fasel *et al.*, "The Universe at Extreme Scale: Multi-Petaflop Sky Simulation on the BG/Q," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 4.

[5] A. Canning, J. Shalf, N. Wright, S. Anderson, and M. Gajbe, "A Hybrid MPI/OpenMP 3D FFT for Plane Wave First-principles Materials Science Codes," in *Proceedings of CSC12 Conference*, 2012.

[6] J. M. Levesque, R. Sankaran, and R. Grout, "Hybridizing S3D into an Exascale Application Using OpenACC: An Approach for Moving to Multi-petaflops and Beyond," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, pp. 15:1–15:11.

[7] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Scaling Hypre's Multigrid Solvers to 100,000 Cores," in *High-Performance Scientific Computing*. Springer, 2012, pp. 261–279.

[8] G. Krawezik, "Performance Comparison of MPI and Three OpenMP Programming Styles on Shared Memory Multiprocessors," in *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 2003, pp. 118–127.

[9] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2010 18th*, 2010, pp. 180–186.

[10] B. Goglin, "Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc)," in *International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2014, pp. 74–81.

[11] F. Wende, T. Steinke, and A. Reinefeld, "The Impact of Process Placement and Oversubscription on Application Performance: A Case Study for Exascale Computing," in *Proceedings of the 3rd International Conference on Exascale Applications and Software*, ser. EASC '15, 2015, pp. 13–18.

[12] S. K. Gutiérrez, "The QUO Runtime Library," Jan 2013, Los Alamos National Laboratory LA-CC-13-076. [Online]. Available: https://github.com/lanl/libquo

[13] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[14] A. Arnold, O. Lenz, S. Kesselheim, R. Weeber, F. Fahrenberger, D. Roehm, P. Košovan, and C. Holm, "ESPResSo 3.1 — Molecular Dynamics Software for Coarse-Grained Models," in *Meshfree Methods for Partial Differential Equations VI*, ser. Lecture Notes in Computational Science and Engineering, M. Griebel and M. A. Schweitzer, Eds., vol. 89. Springer, 2013, pp. 1–23.

[15] R. Thakur and W. Gropp, "Test Suite for Evaluating performance of MPI Implementations That Support MPI_THREAD_MULTIPLE," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2007, pp. 46–55.

[16] R. Rabenseifner, "Hybrid Parallel Programming: Performance Problems and Chances," in *Proceedings of the 45th Cray User Group Conference, Ohio*, 2003, pp. 12–16.

[17] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes," in *Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2009 17th*, 2009, pp. 427–436.

[18] E. Chow and D. Hysom, "Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters," *LLNL, Tech. Rep. UCRL-JC-143957*, 2001.

[19] N. Drosinos and N. Koziris, "Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004.

[20] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A Unified Parallel Runtime for Clusters of NUMA Machines," in *European Conference on Parallel Processing*. Springer, 2008, pp. 78–88.

[21] C. Huang, O. Lawlor, and L. V. Kale, "Adaptive MPI," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2003, pp. 306–322.