STAPL-RTS: An Application Driven Runtime System

Ioannis Papadopoulos
ipapadop@cse.tamu.edu
Texas A&M University
College Station, Texas 77843

Nathan Thomas
nthomas@cse.tamu.edu
Texas A&M University
College Station, Texas 77843

Adam Fidel
fidel@cse.tamu.edu
Texas A&M University
College Station, Texas 77843

Nancy M. Amato
amato@cse.tamu.edu
Texas A&M University
College Station, Texas 77843

Lawrence Rauchwerger
rwerger@cse.tamu.edu
Texas A&M University
College Station, Texas 77843

ABSTRACT

Modern HPC systems are growing in complexity, as they move towards deeper memory hierarchies and increasing use of computational heterogeneity via GPUs or other accelerators. When developing applications for these platforms, programmers are faced with two bad choices. On one hand, they can explicitly manage all machine resources, writing programs decorated with low level primitives from multiple APIs (e.g. Hybrid MPI / OpenMP applications). Though seemingly necessary for efficient execution, it is an inherently non-scalable way to write software. Without a *separation of concerns*, only small programs written by expert developers actually achieve this efficiency. Furthermore, the implementations are rigid, difficult to extend, and not portable. Alternatively, users can adopt higher level programming environments to abstract away these concerns. Extensibility and portability, however, often come at the cost of lost performance. The mapping of a user's application onto the system now occurs without the contextual information that was immediately available in the more coupled approach.

In this paper, we describe a framework for the transfer of high level, application semantic knowledge into lower levels of the software stack at an *appropriate level of abstraction*. Using the STAPL library, we demonstrate how this information guides important decisions in the runtime system (STAPL-RTS), such as multi-protocol communication coordination and request aggregation. Through examples, we show how generic programming idioms already known to C++ programmers are used to annotate calls and increase performance.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS'15, June 8–11, 2015, Newport Beach, CA, USA.
Copyright © 2015 ACM 978-1-4503-3559-1/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2751205.2751233.

Keywords

Parallel Programming; Data flow; Runtime Systems; Application Driven Optimizations; Distributed Memory; Shared Memory; Remote Method Invocation

1. INTRODUCTION

The current state of the art in high performance computing (HPC) is a distributed memory machine comprised of nodes with accelerators and multiple processor sockets, each with a multi-core chip. Application development for these platforms is usually evolutionary: a scalable, distributed programming model (usually MPI [37]) is used for the initial implementation, with the memory hierarchy largely ignored. To increase performance, the implementation is extended with another library (e.g., OpenMP [30]), with threading for finer grain parallelism and shared memory with explicit synchronization to replace communication between processing elements. Writing such programs decorated with primitives from multiple low level APIs is an inherently non-scalable way to write software. Without a *separation of concerns*, only small programs written by expert developers actually achieve greater efficiency. The implementations are also rigid, difficult to extend, and not portable.

This lack of abstraction clearly detracts from code reuse and program composability. However, developers are often faced with no other choice if they wish to gain even some fraction of the peak performance modern systems offer. Efficiently mapping applications to such architectures requires semantic information that is usually lost when higher level programming models are used.

In this paper, we describe how user-level information is transferred to the runtime system of STAPL [10], a generic library of components for parallel program composition. One of the key design goals of STAPL is portable performance: users must be able to write one version of the code that has good performance on different systems with minimal per-platform effort. The layered component architecture of the library supports this objective, with each component responsible for abstracting some area of concern in parallel programming, such as data distribution, computation specification, work scheduling, and communication. Key to obtaining performance is a *transfer of contextual information* between these components, while still maintaining the proper abstractions necessary for software reuse.

The *STAPL Runtime System* (STAPL-RTS) presents a *unified interface* for both intra-node and inter-node communi-

cation to support performance portability. Internally the *mixed-mode* implementation uses both standard shared and distributed memory communication protocols when appropriate. For scalability and correctness, we employ a *distributed Remote Method Invocation* (RMI) model.

Each processing element together with a logical address space forms an isolated computational unit called a *location*. Hence, parameters to RMIs are passed by value, maintaining strict copy semantics with no user-visible sharing. This approach provides safety to the user by guarding against data races. However, as with other features of higher level languages, it can introduce runtime overhead, in this case from excessive copying of large data structures. We show in this paper how *copy elision* (i.e. removing unnecessary copying of objects) can eliminate this performance penalty, via simple annotations inserted by STAPL based on information from higher levels of the software stack.

This paper makes the following contributions:

- **Transfer of application semantics to the runtime.** We employ annotations based on common programming idioms. As STAPL is implemented in C++11 [38], the annotations are similar to C++ standard library interfaces.

- **Copy removal via move semantics and immutable sharing**. To demonstrate application driven optimization, we transparently avoid copies usually incurred when maintaining isolation of computational activities. We employ the commonly known idioms of move semantics [38] and immutable sharing [18], leveraging shared memory for communication between activities whenever possible. STAPL programs are expressed as task dependence graphs, with consumer tasks receiving read-only access to produced values. We use this graph representation to transparently insert annotations whenever possible: tasks with single consumers can direct the runtime to *move* their copy directly to the producer, which it will do if the tasks exist in shared memory. Tasks with multiple consumers can request *immutable references* to the value be transmitted to other locations where these successors exist.

- **Algorithm driven request aggregation.** As another example of application driven runtime optimization, we tune the aggregation of RMIs, an optimization that has been shown to be important for fine-grained asynchronous messaging models [24,33,41]. We create ad-hoc communication channels to efficiently aggregate sequences of RMIs sharing common and constant parameters such as destination and target method. As we will show, the technique can specify custom consistency models for collections of requests that are logically associated with a given computational activity. This technique can have a dramatic effect on application performance, as demonstrated using a common graph traversal algorithm.

2. STAPL OVERVIEW

The *Standard Template Adaptive Parallel Library* (STAPL) [10] is a framework developed in C++ for parallel programming. Both STAPL and the runtime system presented in this paper are libraries, requiring only a C++ compiler (e.g., `gcc`) and established communication libraries such as MPI. An overview of its major components are presented in Figure 1. The library's generic design is based on that of the C++ Standard Template Library (STL) [28], extended and modified for parallel programming.

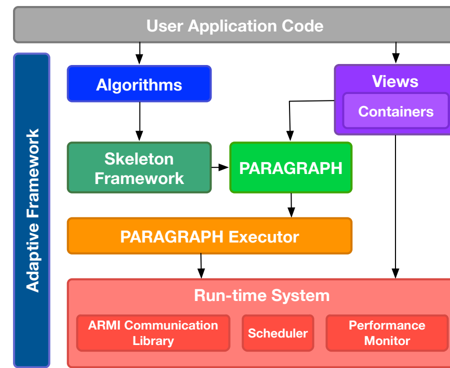


Figure 1: STAPL Components

STAPL provides *parallel algorithms* and *distributed data structures* [20,39] with interfaces similar to the STL. Instead of using iterators, algorithms are written with *views* [9] that decouple the container interfaces from the underlying storage. The *skeletons framework* [43] allows the user to express an application as a composition of parallel patterns.

Algorithmic skeletons are instantiated at runtime as task dependence graphs by the `PARAGRAPH`, STAPL's data flow engine. The `PARAGRAPH` enforces the specified task dependencies and is responsible for the transmission of intermediate values between tasks. When consumer tasks execute on different locations than the producer, the `PARAGRAPH` uses STAPL-RTS primitives for communication. In Section 4.1.1 we discuss how the `PARAGRAPH` annotates these RMI invocations for copy elision, based on information gathered by the program's task dependence graph, without any changes to the program's skeleton specification.

3. THE STAPL RUNTIME SYSTEM

The STAPL-RTS abstracts the platform and its resources, providing a uniform interface for all communication in the library and applications built with it. It provides an SPMD execution model with task parallelism capabilities. Locations only have access to their own address space and communicate with other locations using *Remote Method Invocations* (RMIs) on shared objects. This abstraction of a virtual distributed, parallel machine helps STAPL support general nested parallelism. Similar to the `PARAGRAPH` discussed earlier, containers are distributed objects (i.e., `p_objects`) which use RMIs to read and write elements. In this section, we briefly describe the major components and the interface for RMI invocations. We then describe the STAPL-RTS execution model and discuss design decisions that motivate the application driven optimizations that follow in Section 4.

3.1 Component Overview

The runtime system features a highly modular design, depicted in Figure 2[1], that allows it to be customized and tuned as needed for different platforms. The primary public APIs of STAPL-RTS are briefly described below:

[1]White text signifies components that are user accessible, black text marks components for internal use only.

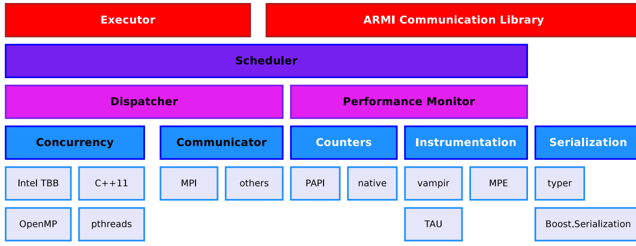


Figure 2: STAPL-RTS components.

ARMI. *Adaptive Remote Method Invocation* (ARMI) [33] provides primitives for registering `p_objects` and invoking RMIs on them. These RMIs allow the asynchronous transfer of both data and work on the system. ARMI also makes use of *future* and *promise* objects [5] to allow the asynchronous return of values from RMIs.

Executor. The EXECUTOR allows the `PARAGRAPH` to schedule runnable tasks for execution with associated scheduling information. It also has work-stealing capabilities [16].

Counters and **Instrumentation**. The STAPL-RTS offers a high-level interface to the native counters available, such as Linux timers or PAPI [2] as well as an *instrumentation* component for tracing and profiling capabilities through the integration of libraries such as TAU [34] and MPE [42].

Serialization. The *serialization* component is used by other components to marshal objects for communication or storage. Packing and unpacking is automatic for simple types, such as empty classes, primitive types and plain old data structures (PODs). For more complex objects, users have to provide a simple definition of the members of each class. It supports integration with other marshaling libraries, such as Boost.Serialization [1].

3.2 The ARMI Interface

An overview of the most commonly used functions from ARMI is presented in Table 1. All primitives are non-blocking, apart from `sync_rmi()`, `rmi_fence()` and `rmi_barrier()`. Primitives provided include standard point-to-point RMIs as well as collective versions for both the case in which all locations participate in the operation and for which a single location initiates the call (one-sided collectives). In Figure 3 we give a simple example of ARMI usage; additional primitives will be discussed as they are used in Section 4.

```
1  struct A : public p_object {
2   int m_value;
3   void write(int t) { m_value = t; }
4   int read() const { return m_value; }
5  };
6
7  foo(...) {
8    A a;
9    auto h = a.get_rmi_handle();
10   int t = 5;
11   async_rmi(1, h, &A::write, t);
12   t = 6;
13   future<int> f = opaque_rmi(1, h, &A::read);
14   int y = f.get();
15 }
```

Figure 3: Basic Usage of ARMI Primitives.

Primitive	Description
One-Sided Primitives	
void async_rmi(dest, h, f, args ...)	Issues an RMI that calls the function `f` of the `p_object` associated with the `rmi_handle h` on location `dest` with the given arguments, ignoring the return value. Synchronization calls or other RMI requests that do not ignore the return value can be used to guarantee its completion.
future<Rtn> opaque_rmi(dest, h, f, args...)	Returns a `future` object for retrieving the return value of the function.
Rtn sync_rmi(dest, h, f, args ...)	Issues the RMI and waits for the return value (blocking primitive).
Collective Primitives	
futures<Rtn> allgather_rmi(h, f, args ...)	The function is called on all locations and the `futures` object is used to retrieve the return values.
future<Rtn> allreduce_rmi(op, h, f, args ...)	The `future` is used to retrieve the reduction of the return values of `f` from each location.
future<Rtn> broadcast_rmi(h, f, args...)	The caller (root) location calls the function and broadcasts the return value to all other locations. Non-root locations have to call `broadcast_rmi(root, f)` to complete the collective operation.
Synchronization Primitives	
void rmi_fence()	Guarantees that all invoked RMI requests have been processed using an algorithm similar to [36].
void rmi_barrier()	Performs a barrier operation.
void p_object::advance_epoch()	Advances the epoch of the `p_object`, as well as the epoch of the location. It can be used for synchronization without communication, avoiding the `rmi_fence()` or `rmi_barrier()` primitives.

Table 1: ARMI Primitives

In Figure 3, function `foo()` is executing on a given location which wishes to communicate with location 1. The shared `p_object a` is accessed through a handle `h`, which represents the distributed object with a representative on the destination. The corresponding instance of `a` on location 1 is updated via a call to `A::write()`. Note that pass by value semantics guarantee that the callee sees 5 and not 6. Also, assuming that no other locations send updates to location 1, y will be set to 6 since the ordering of RMI invocations from a single source is enforced by default.

3.3 STAPL-RTS Execution Model

We describe several aspects of the execution model of the STAPL-RTS, motivating design decisions and pinpointing opportunities for application driven optimization.

Execution Environment. A STAPL application is always implicitly parallel and executes on a number of locations. Each location has an isolated, virtual address space which is not directly accessible by other locations. When a location wishes to modify or read a remote location's memory, the work must be expressed via RMIs on distributed `p_objects`, *even if the two locations reside in shared memory.* This design has the following ramifications:

- **Data Races Cannot Occur.** With only one processing element able to directly access memory and RMI atomicity guaranteed by the runtime system, users do not have the ability to create data races as is usually possible in shared memory parallel execution models.

- **Isolation causes copying.** One cost of the added safety is object copying between locations, even if they share a common address space. Maintaining isolation means values returned from RMIs between locations must be copied. We discuss in Section 4.1 how to minimize this overhead when in shared memory.

Asynchronous Communication Primitives. Asynchrony allows us to minimize the effects of high latency by enabling communication and computation overlapping. We further desire to minimize any state associated with the RMI from the initiating location after invoking the non-blocking RMI, allowing the location to proceed with other, potentially unrelated tasks while it awaits any return. Also, in Section 4.1.4 we show how asynchronous RMI return values are handled without requiring blocking, allowing users to move on to another computation while awaiting said value.

RMI Argument Copy Semantics. We enforce pass-by-value semantics for all arguments passed to an RMI. A private copy of any argument passed to a remote function call is presented to the receiver; any mutation on the argument either at the sender or at the receiver will not be visible to the other. Again, if this is done without high level information, the runtime may introduce unnecessary copies to enforce pass-by-value semantics.

Causal RMI Consistency Model. In order to present a coherent model, our RMIs follow a *causal consistency model*. A happened-before relationship is established between RMIs that are invoked from the same source location to the same destination location if they are issued in the same context without requiring extra synchronization. However, these ordering guarantees may be stricter than required by some algorithms, making it a good candidate for application driven optimization in the STAPL-RTS.

4. APPLICATION DRIVEN OPTIMIZATION

We now give examples of how high level information is transferred from STAPL programs into the runtime system to guide optimization. We begin with `PARAGRAPH` directed copy elision between locations in shared memory. We then discuss communication optimization based on RMI aggregation and consistency model tuning. The information is provided at an appropriate level of abstraction (i.e., they need not be aware of how and if STAPL-RTS uses this information) through a well known programming idiom, derived from a standard C++ language feature or library interface.

4.1 Copy Elision in Shared Memory

Copy semantics simplify the reasoning about parallel programs, as they remove potential side-effects. However they can introduce significant runtime overhead. We relax our implementation of copy semantics with assistance from the STAPL `PARAGRAPH`. We describe three RMI annotations that allow STAPL-RTS to remove copies. They include transfer via `move`, return storage specification via `promise` and `future` objects, and the use of shared, immutable data references. We first describe how application contextual information flows to the `PARAGRAPH` to guide copy elision and then discuss the annotations as well as their implementation.

4.1.1 *PARAGRAPH Direction of Copy Elision*

Copy elision annotations are not inserted by STAPL application programmers. Instead the elision is directed by the `PARAGRAPH`, and Figure 4 depicts this process. First an application writer employs an algorithmic skeleton, which the skeleton framework uses to generate a task dependence graph specification. At runtime, this graph is instantiated by the `PARAGRAPH` and mapped onto a set of locations for execution. The `PARAGRAPH` also performs an inspection of the graph to detect where copy elision can be used. In Figure 4 the result of the map operation on location 1 can be transferred (i.e., moved) to the reduction task on location 0, as it is the only consumer of the value. The `PARAGRAPH` uses the following set of rules to identify elision opportunities:

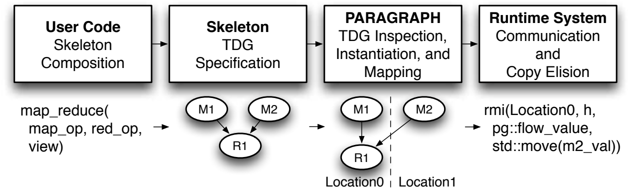


Figure 4: Copy Elision in STAPL.

```
1 void produce() {
2   std::vector<int> v(N);
3   ... // populate v
4   async_rmi(dest, h, obj::consume, v);
5 }
```

(a) RMI invocation with copy of `v`.

```
1 void produce(...) {
2   std::vector<int> v(N);
3   ... // populate v
4   async_rmi(dest, h, obj::consume, std::move(v));
5 }
```

(b) RMI invocation with move of `v`.

Figure 5: Move Semantics for RMI Arguments

- `move` **annotation**. If a task has single consumer and it is on a remote location (i.e., different than where the task executes), pass the value to `async_rmi` via `std::move`.
- `immutable_shared` **annotation**. If a task has multiple consumers and at least one is on a remote location, use an immutable shared reference. The reference is passed to associated RMIs and also used to service local consumers.
- **No annotation**. If all consumers are on the producer's location, the value is managed locally with no RMIs.

4.1.2 *Using Moves for RMI Parameter Passing*

Consider the code in Figure 5(a) which calls `async_rmi()`. A location executes function `produce()` that creates a vector and sends it to another location via RMI. In this case, the copy of the vector parameter into the runtime is unnecessary. The source location produces the value solely for consumption at the destination location. This is an *object transfer* pattern present in many parallel algorithms (e.g., reductions). This type of value transfer is also desirable in sequential computing. C++11 [38] addresses this problem with language support for *rvalue references* and an associated library function `std::move()`.

The STAPL-RTS supports the direct use of these move semantics with RMI parameter passing, so that the unneces-

```
1 foo(...) {
2   auto t = make_immutable_shared<T>(...);
3   async_rmi(1, h, &A::put, t);
4   async_rmi(2, h, &A::put, t);
5   T const& ref = t.get();
6 }
```

(a) Example `immutable_shared` usage.

```
1 t = ...;
2 async_rmi(1, h, &A::put, immutable(t));
3 rmi_fence();
4 t = ...;
```

(b) Example `immutable` usage.

Figure 6: Immutable object sharing in STAPL-RTS.

sary copies can be completely avoided. The trivially modified code in Figure 5(b) has been annotated to express the transfer of `v`. The parameter is passed without any copying *when the source and destination reside in the same address space* and is presented to `consume_value()` as an rvalue reference, that is as an `std::vector<int>&&`. During execution, the parameter is moved from user space into the runtime, serialization is avoided, and control bits are inserted into the RMI request to forward the rvalue reference to the callee.

4.1.3 Immutable Object Sharing

There are times that basic *data transfer* between locations is insufficient. If there is still a local consumer of the value to be transmitted remotely, we can employ *immutable data sharing* to avoid overhead while still preserving copy semantics. This admittedly does not cover all cases (i.e., if the receiver wants to mutate the value, they must still copy it), but when it can be used, it gives similar savings as the zero-copy data transfers discussed in the previous two sections. We currently offer two variations of immutable sharing:

Permanently immutable objects. Values placed in an immutable wrapper via `make_shared_immutable()` are guarded against mutation for the remainder of their lifetime. This STAPL-RTS primitive mimics the behavior of a `std::shared_ptr<const T>` and is used to safely share values between locations in shared memory. When the destination location resides in another address space, a new copy is initialized there to back the immutable wrapper. In each address space, the underlying copy is deleted when the last reference is deleted, using standard reference counting.

Figure 6(a) depicts an example of permanently immutable sharing. Assume that location 1 resides in the same address space as the location executing `foo()`, while location 2 does not. Location 1 shares a copy of `t` with the lifetime managed by STAPL-RTS. When `foo` exits and references on locations 1 are destroyed, the copy is destroyed. Location 2 receives a wrapper to its own copy which is read and subsequently shared, if desired, with other locations.

Temporarily immutable objects. This annotation, demonstrated in Figure 6(b), allows the caller to regain mutability rights of the object after the next synchronization point. Objects are tagged using the `immutable()` function. A reference to such an object can be given to a destination location in shared memory, instead of copying it. Using the `immutable` tag, the caller guarantees that `t` will not be updated until after the `rmi_fence()` collective synchronization call. Afterwards the variable can be safely modified.

```
1  send_request(...) {
2    future<T> f = opaque_rmi(0, h, &A::get_value);
3    return f;
4  }
5
6  process_request(T& t, future<T> f) {
7    if (f.valid()) {
8      t = f.get();
9      return true;
10   }
11   return false;
12 }
13
14 stapl_main(...) {
15   T t;
16   future<T> f = send_request();
17   while (!process_request(t,f))
18   { ... // perform other work }
19   foo(t); // use t;
20 }
```

(a) Example `stapl::future` and `stapl::promise` usage.

```
1 send_request(...) {
2    future<T> f1 = opaque_rmi(1, h, &A::get_value)
3    f1.then([](future<T> f2) { foo(f2.get()); });
4 }
5
6 stapl_main(...) {
7   send_request();
8   ... // proceed with other work
9 };
```

(b) Example `stapl::future::then()` usage.

Figure 7: Asynchronous value retrieval in STAPL-RTS.

4.1.4 Futures for RMI Return Values

While the two elision idioms described above adequately address the STAPL-RTS equivalent of *put operations* via parameter passing, for asynchronous *get operations* we draw inspiration from two other C++ STL primitives, `std::future` and `std::promise`. A *future* [5] is a mechanism to retrieve the result of an asynchronous primitive that does not ignore the result of the invoked function, e.g. `opaque_rmi()`. The *promise* is a placeholder for an incoming value, which can be set at the end of complex, multi-hop communication.

While C++ versions are for shared memory, our implementation provides similar semantics transparently in distributed memory without any additional intervention from users. The promise/future provides a standard idiom to facilitate zero copy *gets* and delegate responsibility for receiving the return value from the RMI to code outside the calling context. One example usage is shown Figure 7(a).

This trivial example shows how a return value from an RMI can be handled outside the context of the RMI invocation. A different computational activity can occur while waiting for the internal STAPL-RTS `promise` associated with the `future` to be fulfilled. We also support continuations on future objects through the `future::then()` function [12], an extension that has been proposed for C++17 [19]. Again, we follow the proposed interface, but provide our own implementation that provides a uniform interface for both shared and distributed memory. Together with a lambda expression, this feature is used to refine the previous example as shown in Figure 7(b). In this case, the consuming function of the RMI return value is specified at the RMI call site, and

```
for (int idx = ...)
  async_rmi(dest, handle, A::set_element,
            idx, rand());
```

(a) Default Request Aggregation.

```
auto tunnel = bind(async_rmi, dest, handle,
                   A::set_element, _1, _2);
for (int idx = ...)
  tunnel(idx, rand());
```

(b) Partial Function Evaluation of `async_rmi`.

Figure 8: Application Customized Aggregation in STAPL-RTS.

will be called by STAPL-RTS when the corresponding promise is fulfilled. Other local computation proceeds immediately after the initial RMI request is made.

The `PARAGRAPH` makes use of the `future` and `promise` primitives to easily spawn nested parallel computations from a source location to a set of remote locations. When the nested computation completes, the initiating location asynchronously receives notification of completion from an event handler registered with the STAPL-RTS via `future::then()`.

4.2 Application Driven Request Aggregation

As is usually the case with asynchronous communication models, STAPL encourages fine grain communication. Previous work [24,33,41] has shown aggregating these requests generally leads to overall better performance. In STAPL-RTS, we aggregate multiple RMI requests to the same destination location in the same outgoing buffer. We further enhance this mechanism by implementing *request combining*, a lightweight compression technique for requests that have the same triplet of target `p_object`, function and destination as the previous requests in the buffer. If this triplet is the same, then we need only append the arguments of the request to the aggregation buffer.

Consider the code in Figure 8(a) which updates a sequence of values in a remote object with random values. If the STAPL-RTS were to generate an MPI request for every `async_rmi` invocation in this tight loop, performance would suffer as the overhead of request transmission would greatly outweigh the cost of the requested updates. In this case, however, the STAPL-RTS is free to employ not only basic aggregation but combining as well, without violating the causal consistency guarantee of the execution model.

Note however in this case there is information trivially available to the user that would aid the runtime in this activity. The fact that the object handle, destination location and target method remain constant is immediately clear in the calling context. Using *partial function evaluation*, a common generic programming operation, we can fix one or more arguments of the STAPL-RTS primitives such as `async_rmi`, creating a new function with reduced arity. This new function contains typing information about which parameters have been fixed. To accomplish this, we can use a `bind` function with an interface similar to that of the C++ STL. The operation creates a custom communication channel as shown in Figure 8(b) based solely on algorithm level information. Using this RMI *tunnel* has the following effects:

Relaxed request consistency. Tunnels define a new logical route to destination location with ordering consistency guarantees independent of the default route (the atomicity of all RMIs is maintained). A tunnel defined with only a bound location maintains the same basic causal consistency policy as previously described.

Less runtime overhead and more efficient aggregation. By binding additional arguments during the partial evaluation, we reduce redundancy in the message; only a single copy of the bound parameter is stored in the aggregated message instead of a copy for every RMI. Other optimizations are enabled by different combinations of bound parameters. For example binding both the object handle and target function enables combining at compile time (Section 3.3), eliminating the overhead of runtime detection.

Though not necessary for the example shown in Figure 8(a), dedicated per method tunnels enable greater use of combining in some instances. Consider the case where a small number of non-combinable RMIs are interspersed in an otherwise homogeneous sequence of RMI invocations. By creating a tunnel for the homogeneous requests, the other requests do not interfere with the combining operation.

One use of tunnels in STAPL are in graph traversals, where a vertex *visitor* function passed to the algorithm is repeatedly applied on vertices throughout the `pGraph` data structure. When graph edges cross location boundaries, RMIs are issued to complete the visitation. The algorithm specifies a set of tunnels for these fine grain method invocations through this interface. As we show in Section 5.3, this high level annotation can have a dramatic effect on performance and scalability.

5. EXPERIMENTAL EVALUATION

Our experiments were carried out on three different systems. The code was compiled with maximum optimization levels (`-DNDEBUG -O3`). Each experiment has been repeated 32 times and we present the average along with a 95% confidence interval using the t-distribution.

CRAY-XK7**.** This is our department's Cray XK7m-200 system which consists of twenty-four compute nodes with AMD Opteron 6272 Interlagos 16-core processors at 2.1 GHz. Twelve of the nodes are single socket with 32 GB of memory, and the remaining twelve are dual socket nodes with 64 GB. Our benchmark code has been compiled with `gcc` 4.8.1 and we configured the STAPL-RTS with the OpenMP-based concurrency back-end with each location mapped to one OpenMP thread, pinned to one core.

IBM-BG/Q**.** This is an IBM BG/Q system available at Lawrence Livermore National Laboratory. IBM-BG/Q has 24,576 nodes, with each node populated by a 16-core IBM PowerPC A2 processor clocked at 1.6 GHz and 16 GB of memory. The compiler used was `gcc` 4.7.2, and we used the C++11-thread based multithreaded backend.

X86-CLUSTER**.** This machine is an `x86`-based commodity cluster that consists of 311 nodes with different processor and memory configurations. The slice of the system that we used for our experiments is 128 nodes. Each node has two AMD Opteron 2350 2.5 GHz processors, with each processor having 4 cores, for a total of 8 cores and 32 GB per node. We used `gcc` 4.8.2 and the C++11 thread-backed backend.

5.1 Copy Elision in K-means Clustering

K-means clustering [17] is a widely used data mining algorithm. Given a set of vectors in an n-dimensional space, the algorithm assigns vectors which are similar to one an-

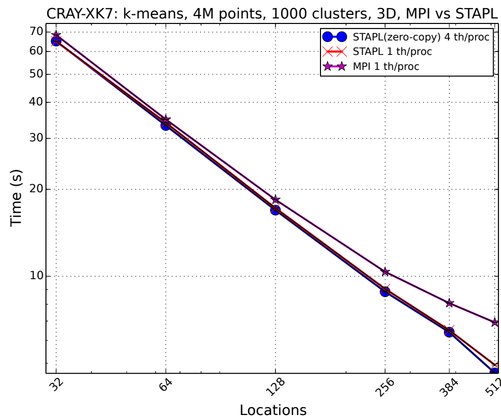


Figure 9: K-means algorithm with 4 M points, 1000 clusters in 3 D space on CRAY-XK7.

other to a specific cluster. The "K" refers to the number of clusters, which is specified by the user, at the start of the algorithm. The "means" refers to the computation for associating the vectors. Each cluster is represented by a single point in the space, which is referred to as a *cluster means* or *cluster centroid*. Dhillon and Modha [15] present a sequential and analogous parallel implementation of k-means. The parallel algorithm is implemented using MPI.

We implemented the Dhillon and Modha MPI version of k-means in C++ and then created a STAPL implementation of the same algorithm. It employs an algorithmic skeleton performing a `map` operation followed by an `all_reduce`. Binary reduction tasks use moves on one of the inputs to colocate data for the operation. The broadcast portion of the allreduce operation uses shared immutability to avoid unnecessary copies during dissemination of new cluster centroids. These optimizations are enabled by the PARAGRAPH using the rules outlined in 4.1.1.

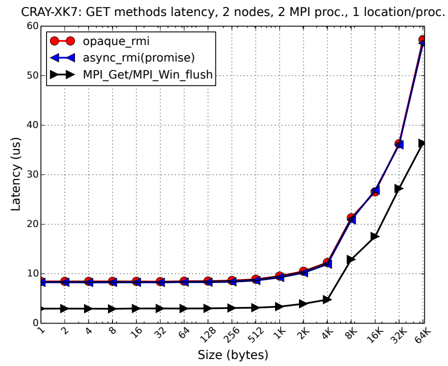
The scalability of the STAPL version (`STAPL 1th/proc`) as shown in Figure 9 surpasses that of the MPI implementation, due to other optimizations besides copy elision. Despite being primarily a computation kernel, the mixed-mode execution with copy elision (`STAPL(zero-copy) 4th/proc`) sees gains of up to 6.2% over the basic STAPL implementation.

5.2 Asynchronous Return Values

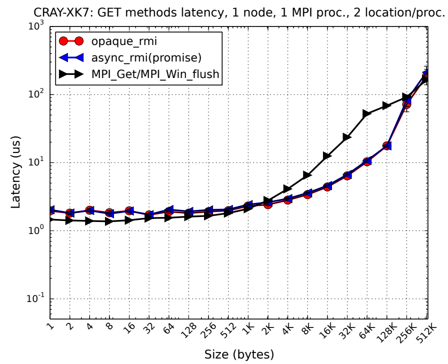
RMIs offer additional flexibility compared to MPI for value retrieval. MPI Remote Memory Access (RMA) [27] use is complex, requiring explicit memory registration and synchronizations. RMIs expose a simple, high level interface which allows one to either wait for values or pass them to a continuation via `future::then()`, with data transfer details managed by the STAPL-RTS.

Figure 10 presents the latency of our primitives employing `future`s and `promise`s (Section 4.1.4). We compare against one-sided MPI with `MPI_Get()`/`MPI_Win_flush()`[2] under distributed and shared memory. MPI windows are created through `MPI_Win_create_dynamic()`, as it provides the most flexible form of RMA memory registration for non-trivial applications [7].

[2]This was the best performing combination on our system.



(a) Distributed memory.



(b) Shared memory (log y-axis).

Figure 10: `opaque_rmi()`/`async_rmi(promise)` latency against One-Sided MPI.

Both our methods (`opaque_rmi()` and `async_rmi()` with `promise`) have similar latency. They are competitive with MPI in distributed memory. However as the object size increases past 64 KB, serialization begins to noticeably affect perceived latency. In shared memory, especially for medium object sizes (2 KB - 256 KB), we outperform MPI, as we can automatically elide one memory copy that MPI has by performing in-place construction of the object in the receiver's address space. MPI starts to outperform us after 512 KB.

5.3 Custom Aggregation in Graph Algorithms

We evaluate the STAPL-RTS using a parallel connected components (CC) graph algorithm. The algorithm computes the connected components – i.e., the subgraph wherein

```
1 for (auto&& u : neighbors(v))
2   async_rmi(location_of(u), handle,
      Graph::visit<cc_visitor>, cc_visitor(v.id()),
      u);
```

(a) Default Request Aggregation.

```
1 auto tunnel = bind(async_rmi, _1, handle,
      Graph::visit<cc_visitor>, _2, _3);
2 for (auto&& u : neighbors(v))
3   tunnel(location_of(u), cc_visitor(v.id()), u);
```

(b) Partial Evaluation of `async_rmi`.

Figure 11: Customized Request aggregation for connected components.

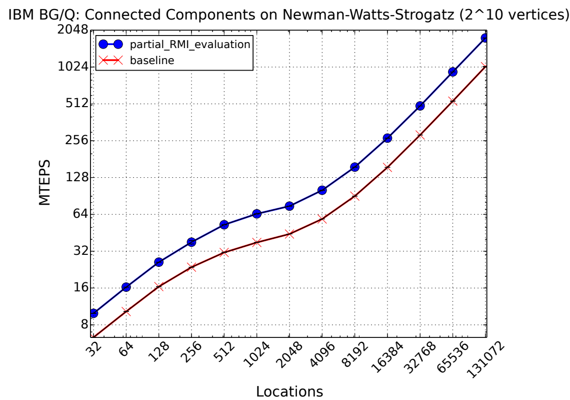


Figure 12: Connected Components on IBM-BG/Q**.**

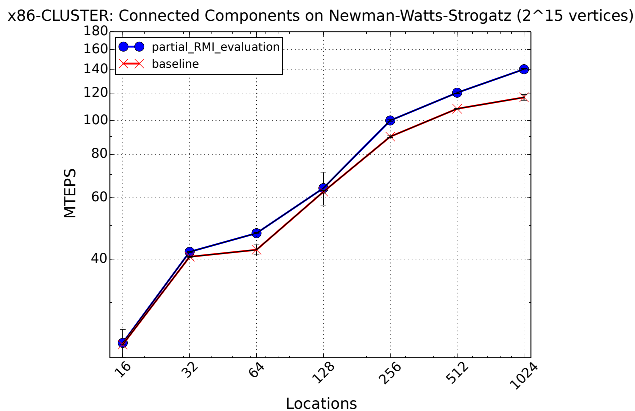


Figure 13: Connected Components on x86-cluster**.**

any two vertices in the subgraph can be connected through some path – for each vertex, and the ID representing the component is assigned to the vertex. It is a label-propagation algorithm similar to the work presented in [25], wherein nodes set their CC ID to the lowest CC ID of their neighbors iteratively for k rounds. Connected components is widely used to study the connectivity and basic topology of graphs.

In the standard expression of the algorithm, each vertex in parallel visits all other vertices in its neighborhood to propagate its ID as a candidate CC ID. In Figure 11(a), we achieve this by issuing `async_rmi`s to the locations of all neighbor vertices to apply the visitation function computing the CC. At the algorithm's level, both the handle of the graph data-structure and the method to apply visitor functions is constant, so we can form a tunnel to aid combining based aggregation of these requests (Figure 11(b)).

Figure 12 evaluates the algorithm's performance in terms of throughput (millions of traversed edges per second or MTEPS), both with and without tunneling on up to 131, 072 processors on IBM-BG/Q for a Newman-Watts-Strogatz graph of 2^{10} vertices per core. We see a 1.5× improvement in throughput at lower core counts, which grows to 1.7×, suggesting that tunneling is not only increasing throughput but also improving scalability. Figure 13 shows the performance for the same type of graph with 2^{15} vertices per core on x86-cluster with smaller but still noticeable improvement of about 20% over the approach without tunneling.

6. RELATED WORK

In [23] the authors propose Kanor, a declarative language for writing parallel programs in partitioned address spaces. Users annotate how data flows between address spaces to describe communication. The Kanor source-to-source compiler uses these annotations to do a more informed data-flow analysis and appropriately promote objects to global shared objects in shared memory, achieving zero-copy without explicit synchronization from the user's side, while maintaining the isolation features of distributed memory. The resulting code targets either multithreaded code or MPI code, but not a mix of both. The immutable object support we present is similar to performing the globalization optimization on a variable in Kanor but with the added benefit that it works in mixed-mode as well. A compiler such as Kanor's could easily leverage the immutable object support we offer.

A framework for taking advantage of immutable objects is introduced in [32] for code optimization. The authors describe a set of immutability annotations for Java that can be added to local or member variables. These can be used by the compiler to perform optimizations such as relaxing bounds checking, and load eliminations.

Several papers explore reference and object immutability for type safety reasons, with the potential to enable optimizations using those guarantees. Javari [6] and IGJ [44] extend the Java language with reference and object immutability qualifiers. Using them, they provide *symbolic constants*. While it is mentioned that these qualifiers can enable code optimizations, this opportunity is not explored. The authors of [18] extend the concepts of object and reference immutability to build a type system that offers immutability guarantees for objects, for the purpose of exploiting it in parallel execution. However, the paper does not expand on the performance implications. The presented immutable object support is similar to this work, but our focus is exploiting shared memory, rather than enforcing type safety.

There have been previous attempts to provide a unified communication model. Treadmarks [3] and Intel Cluster OpenMP attempted to expand shared memory models to distributed memory. While novel and popular at the time, such shared memory approaches suffer inherent scalability issues which make them infeasible for large distributed systems. Additionally, it has been suggested that MPI should become aware of shared memory through the use of the RMA functionality [22], but this has yet to be approved in the MPI specification. However, the use of all these primitives must be explicitly set up and managed by the user, making it effectively a multi-protocol approach.

MPI implementations [8, 26] detect intra-node communication and use optimized methods for copying data. While the optimizations take advantage of the node memory hierarchy, data copying is still required between MPI processes.

Hybrid OpenMP+MPI solutions have been used in applications [13, 35] with success. Almost all of these applications have sequences of parallel OpenMP sections followed by sequential sections that perform communication using MPI. The reason for this configuration is that while MPI implementations allow threads to communicate with each other under the `MPI_THREAD_MULTIPLE` mode [27], it has been shown that this negatively affects performance [40].

Habanero-C with MPI (HCMPI) [14] introduces distributed memory communication in Habanero-C. It uses the Habanero task programming model for intra-node computation and

synchronization, while introducing new functions based on MPI for the inter-node equivalents. While HCMPI has better performance than MPI or hybrid MPI+OpenMP, it presents two different programming models to the user. Intra-node, HCMPI uses the Habanero-C interface, while inter-node it relies on a MPI-like message passing interface.

HPX [21] supports hybrid-mode, allowing threads to communicate over distributed memory with asynchronous primitives. Future and promise mechanisms are provided for synchronization and data retrieval. The distributed memory communication employs MPI. In shared memory, it allows arguments passed by reference, significantly reducing communication latency. In distributed memory the arguments are copied, exposing a slightly different view to the user depending on the target of the communication primitive.

Charm++ [31] provides support for hybrid mode, either by having multiple threads per node to process messages or by declaring functions threaded and allowing them to block while waiting for a value to arrive. The user has to be aware of the threading capabilities, as data structures that may be accessed by multiple threads have to be properly protected.

Chapel [11] has been designed to specifically fit multicore distributed systems. As such, they offer support for creating tasks asynchronously either on distributed or shared memory. There is not enough information on how Chapel performs in mixed shared and distributed memory applications. However, shared-memory only performance indicates that Chapel may not adequately optimized [29].

Tpetra [4] is a linear algebra package from Trilinos. Tpetra supports hybrid-mode parallelism through Kokkos; communication in distributed memory uses MPI, while shared memory communication and computation employ various Kokkos backends (Pthreads, OpenMP, Intel TBB, CUDA). Tpetra resembles a hybrid MPI+OpenMP system, as Kokkos tasks cannot communicate through distributed memory.

7. CONCLUSION

In this paper, we have demonstrated that application level information can be passed to lower levels of the software stack in a manner that does not break the software encapsulation of the communicating components yet still enables important runtime optimizations. We believe doing so aids the adoption of higher level parallel programming models which help HPC applications achieve greater reuse and portability.

To prove both the viability and utility of the approach, we use the STAPL programming library, extending the interface of its runtime layer so that higher level components can provide contextual information to it. We show that these additional annotations enable important optimizations, including copy elision and custom request aggregation. We believe that the ideas presented have general applicability beyond STAPL, as other libraries can leverage similar information from their internal program representations to drive optimizations in the runtime system.

8. ACKNOWLEDGMENTS

This research is supported in part by NSF awards CNS-0551685, CCF-0702765, CCF-0833199, CCF-1439145, CCF-1423111, CCF-0830753, IIS-0916053, IIS-0917266, EFRI–1240483, RI-1217991, by NIH NCI R25 CA090301-11, by DOE awards DE-AC02-06CH11357, DE-NA0002376, B575363, by Samsung, IBM, Intel, and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

9. REFERENCES

[1] Boost. `http://www.boost.org/`.

[2] Performance Application Programming Interface. `http://icl.cs.utk.edu/papi/`.

[3] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, Feb. 1996.

[4] C. G. Baker and M. A. Heroux. Tpetra, and the use of generic programming in scientific computing. *Sci. Program.*, 20(2):115–128, Apr. 2012.

[5] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, Aug. 1977.

[6] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *Object-Oriented Prog. Systems, Langs., and Apps. (OOPSLA 2004)*, pages 35–49, Vancouver, BC, Canada, Oct. 2004.

[7] D. Bonachea and J. Duell. Problems with using mpi 1.1 and 2.0 as compilation targets for parallel language implementations. *Int. J. High Perf. Comput. Netw.*, 1(1-3):91–99, Aug. 2004.

[8] D. Buntinas and G. Mercier. Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In *Proc. of the Int. Symp. on Cluster Computing and the Grid*, pages 521–530. IEEE Computer Society, 2006.

[9] A. Buss, A. Fidel, Harshvardhan, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL pView. In *Int. Wkshp. on Langs. and Comps. for Par. Comp. (LCPC)*, Houston, TX, USA, Sept. 2010.

[10] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. STAPL: Standard template adaptive parallel library. In *Proc. Annual Haifa Experimental Systems Conf. (SYSTOR)*, pages 1–10, New York, NY, USA, 2010. ACM.

[11] D. Callahan, B. L. Chamberlain, and H. P. Zima. The cascade high productivity language. In *9th Int. Wkshp. on High-Level Par. Prog. Models and Supportive Environments (HIPS)*, pages 52–60, 2004.

[12] C. Campbell and A. Miller. *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, 1st edition, 2011.

[13] F. Cappello and D. Etiemble. Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In *Proc. of the 2000 ACM/IEEE Conf. on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[14] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan. Integrating asynchronous task parallelism with mpi. In

Proc. Int. Par. and Dist. Proc. Symp. (IPDPS), pages 712–725, May 2013.

[15] I. Dhillon and D. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Large-Scale Par. Data Mining*, volume 1759 of *LNAI*, pages 245–260. Springer-Verlag, 2000.

[16] A. Fidel, S. A. Jacobs, S. Sharma, N. M. Amato, and L. Rauchwerger. Using load balancing to scalably parallelize sampling-based motion planning algorithms. In *Proc. Int. Par. and Dist. Proc. Symp. (IPDPS)*, Phoenix, Arizona, USA, May 2014.

[17] J. Ghosh and A. Liu. K-means. In *The Top Ten Algorithms in Data Mining*. CRC Press, Boca Raton, FL, USA, 2009.

[18] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *Proc. of the ACM Int. Conf. on Object Oriented Prog. Systems Langs. and Apps.*, OOPSLA '12, pages 21–40, New York, NY, USA, 2012. ACM.

[19] D. Gregor, B. Stroustrup, J. Widman, and J. Siek. Improvements to std::future<t> and related apis. technical report n3857. *ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++*, 2014.

[20] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. The stapl parallel graph library. *In Lecture Notes in Comp. Sci. (LNCS)*, pages 46–60. Springer Berlin Heidelberg, 2012.

[21] T. Heller, H. Kaiser, A. Schäfer, and D. Fey. Using hpx and libgeodecomp for scaling hpc applications on heterogeneous supercomputers. In *Proc. of the Wkshp. on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '13, pages 1:1–1:8, New York, NY, USA, 2013. ACM.

[22] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. D. Gropp, V. Kale, and R. Thakur. Mpi + mpi: A new hybrid approach to parallel programming with mpi plus shared memory. *Computing*, 95:1121–1136, 2013.

[23] F. Jiao, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine. Partial globalization of partitioned address spaces for zero-copy communication with shared memory. In *High Perf. Computing (HiPC)*, pages 1–10, Dec 2011.

[24] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, Oct. 1993.

[25] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system-implementation and observations, 2009.

[26] S. N. Labs. Portals Message Passing Interface. `http://www.sandia.gov/Portals`.

[27] MPI forum. MPI: A Message-Passing Interface Standard Version 3.0. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, 2012.

[28] D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide, 2nd Edition*. Addison-Wesley, 2001.

[29] S. Nanz, S. West, and K. S. D. Silveira. B.: Benchmarking usability and performance of multicore languages. In *In: ESEM'13. IEEE Computer Society*, 2013.

[30] OpenMP Architecture Review Board. *OpenMP - C and C++ Application Program Interface*, October 1998. Document DN 004-2229-001.

[31] Parallel Programming Laboratory, University of Illinois at Urbana-Champaign. *The Charm++ Programming Language Manual*. Version 6 (Release 1).

[32] I. Pechtchanski and V. Sarkar. Immutability specification and its applications: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):639–662, Apr. 2005.

[33] S. Saunders and L. Rauchwerger. ARMI: an adaptive, platform independent communication library. In *Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Par. Prog. (PPoPP)*, pages 230–241, San Diego, California, USA, 2003. ACM.

[34] S. S. Shende and A. D. Malony. The tau parallel performance system. *The Int. J. of High Perf. Computing Apps.*, 20:287–331, 2006.

[35] J. Sillero, G. Borrell, J. Jiménez, and R. D. Moser. Hybrid openmp-mpi turbulent boundary layer code over 32k cores. In *Proc. of the 18th European MPI Users' Group Conf. on Recent Advances in the Message Passing Interface*, EuroMPI'11, pages 218–227, Berlin, Heidelberg, 2011. Springer-Verlag.

[36] A. B. Sinha, L. V. Kalé, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm, 1993.

[37] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The complete reference*. MIT Press, Cambridge, MA, 1996.

[38] B. Stroustrup. *The C++ Programming Language*. Addison Wesley Professional, 2013.

[39] G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Container Framework. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, pages 235–246, San Antonio, Texas, USA, 2011.

[40] R. Thakur and W. D. Gropp. Test suite for evaluating performance of multithreaded mpi communication. *Par. Computing*, 35:608–617, Nov. 2008.

[41] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. Am++: A generalized active message framework. In *Proc. of the 19th Int. Conf. on Par. Architectures and Compilation Techniques*, PACT '10, pages 401–410, New York, NY, USA, 2010. ACM.

[42] C. E. Wu et al. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proc. of SC2000: High Perf. Networking and Computing*, Nov. 2000.

[43] M. Zandifar, N. Thomas, N. M. Amato, and L. Rauchwerger. The STAPL skeleton framework. In *Proc. 27th Int. Wkshp. on Langs. and Comps. for Par. Comp. (LCPC), Hillsboro, OR, US*, 2014.

[44] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun, and M. D. Ernst. Object and reference immutability using java generics. In *Proc. of the the 6th Joint Meeting of the European Soft. Eng. Conf. and the ACM SIGSOFT Symp. on The Foundations of Soft. Eng.*, ESEC-FSE '07, pages 75–84, New York, NY, USA, 2007. ACM.