UPC Collectives Library 2.0

George Almási[⋆] Paul Hargrove[†] Ilie Gabriel Tănase[⋆] Yili Zheng[†]

[†] Lawrence Berkeley National Laboratory [⋆] IBM T.J. Watson Research Center

ABSTRACT

Collective communication has been a part of the UPC standard since having been introduced in 2005 with the UPC Specification version 1.2. However, unlike MPI collectives, UPC collectives have never caught on and are rarely used.

In this paper we identify and discuss several fundamental limitations and important missing features in the design of the existing UPC collectives that make them inconvenient to use and unsuitable for performance optimization.

Next, we propose a new, consistent, portable and high performance collectives library that is aimed to augment UPC with a full complement of the collectives used by MPI. Ours is a pure library based approach; we change none of the functions in the existing UPC specification.

We discuss the implementation requirements for this new UPC collectives library, and how our design attempts to minimize the implementation effort by enabling the reuse of existing collective implementations.

1. INTRODUCTION

The notion of collective communication has been popularized by the MPI library [10]. MPI collective primitives are able to express fairly complex combinations of data movement, synchronization and computation operations, and have become well understood tools, or programming patterns, in the arsenal of people writing programs for large parallel machines [7, 15, 16, 14, 8]. Collective communication primitives are anchored in the SPMD programming model, but transcend programming languages. Variations on collective communication have found their way into most modern parallel programming languages, including Partitioned Global Address Space (PGAS) languages [18, 17, 13, 19, 6, 2, 5].

The UPC Specification Version 1.2 [17] contains a number of collective communication operations. It is our contention, and the premise of this paper, that the current standard is flawed and insufficient.

- The current UPC specification lacks equivalents to some of the most popular MPI collectives. Canonical examples include `MPI_Allreduce` (vector reduction with results propagated to every participant) and `MPI_Alltoallv` (personalized communication with variable amount of data).
- The UPC specification lacks the concept of a *team* (or, in MPI parlance, *communicator*. Teams are used to denote subsets of UPC threads to run collectives on. In MPI communicators are of crucial importance: they allow code reuse, and therefore, the development of MPI libraries. We anticipate that a careful design of teams in UPC will help achieve the same purpose.
- The UPC specification is unclear on the correct use of synchronicity flags on collectives. We have anecdotal evidence that the UPC synchronicity flags seem to be hard to master for novice users, and our experience with implementations of the standard suggests that the complexity is not really warranted by the putative improvement in performance that it makes possible.
- The UPC specification has non-blocking barriers, but no other non-blocking collectives. The (fairly complicated) synchronizing flags specification is supposed to allow for overlap of collectives, but this turns out to be awkward for users and difficult for implementors.

The remainder of this work is our proposal to mitigate these problems. In Section 2 we lay out the principles of our approach. Section 3 lays out our new proposed API: type definitions and function prototypes.

No performance results: we do not show any performance results in this paper. Two separate implementations are underway, under the auspices of the authors' host institutions, but neither implementation is ready for in-depth performance testing at press time.

Since we expect implementors to reuse MPI collectives to implement the UPC collectives library 2.0, the

This research was supported in part by the Office of Science of the U.S. Department of Energy under contracts DE-FC03-01ER25509 and DE-AC02-05CH11231, and by the DARPA HPCS Contract HR0011-07-9-0002.

expected performance profile of UPC collectives is not substantially different from that of MPI collectives. ALL-SYNC flags form a notable exception, as their presence may add latency to execution.

2. GENERAL APPROACH

In this Section we describe our approach to define a comprehensive collective communication library compatible with the UPC language.

We do **not** propose to change any existing UPC functionality. We are not changing the semantics of anything that is already defined in the UPC specification 1.2, including collective functionality.

Instead, we propose to define a new library of collective communication primitives with a new API. This API ought to be more familiar to those using collective communication calls in MPI code, and in line with what the MPI forum [9] is proposing for the MPI 3.0 standard [11]. Indeed, we are hoping to enable the reuse of existing MPI collective libraries in UPC.

2.1 Teams

Teams are arbitrary named subsets of UPC threads that collective communication operations can be invoked on. In MPI they are called *communicators*. The current UPC specification does not provide for team based collective communication; we propose to add them to our library.

We start with a small set of functions to manage the life cycle (creation, destruction) of UPC teams. A UPC team is either pre-defined (like `UPC_TEAM_ALL`, meaning all threads) or created with a function like `upccoll_team_split()`, as defined in Section 3.4.

Just like in MPI, team life cycle functions are themselves collectives and have synchronizing properties, so as to avoid situations in which e.g. a broadcast message on a newly created team overtakes the message informing the target UPC thread about the creation of the team.

2.2 Shared arrays and collectives

UPC Spec 1.2 collectives are "single-valued": all UPC threads participating in a collective are required to submit the *same* buffer as argument; failure to do so results in undefined behavior. Furthermore, while arguments to collective calls are all typed as `shared void *` or similar, operation semantics interpret them as shared arrays with finite (but variable) blocking factors.

The obvious limitation of this approach is that only a single shared array can be involved in a collective at a time, frequently necessitating local memory copies before and after the operation. The implicit type cast part of most collective operations also complicates the writing of correct code, since what looks like a contiguous index space to the user is broken up into strided accesses in the collective and vice versa.

Our proposed API corrects this problem by relaxing some of the requirements on user buffers, and moving the interface closer to an "MPI-like feel."

- We allow multi-valued arguments. In other words, different threads can specify different pointers-to-shared in a call to the same collective.

- We require that the buffers submitted in each collective call be affine to the calling thread. Each thread processes data that actually is affine to that thread.

- UPC collectives operate on shared memory buffers. We do not allow UPC collectives to operate on thread-local memory.

- Pointer-to-shared arguments are always interpreted as having indefinite blocking factors (in effect, pointing to contiguous areas of memory).

2.3 Non-blocking collectives

The Berkeley UPC compiler [1] provides an asynchronous memory operator extensions (`upc_memget_async` and friends). This extension is now being proposed as a standard library for UPC [4]. Non-blocking operations have been shown to be effective to improve application performance [3, 12].

We propose similar syntax for non-blocking UPC collectives. The idea is to allow the collective call to return without having completed. The user receives a token for the collective in progress. As in MPI, the token **must** be checked for completion; the collective call completes when the check function `upc_wait` returns `true`.

In order to keep the number of collective primitives down, we use an extension of the `SYNC` flags, described in detail in Section 3.1.2, to specify non-blocking collective calls. We allow three flavors of collective calls: blocking calls, non-blocking calls with explicit synchronization, and non-blocking calls that are synchronized at next invocation of `upccoll_fence()` (see Section 3.3 for a complete description).

2.4 Buffer ownership rules

The MPI standard is very exact about the life cycle and ownership of data buffers passed between users and MPI functions. The principle on which MPI operates is that any buffer passed to MPI cannot be touched by the user (either read or write) until the MPI function is complete. We adopt this principle for buffer ownership for UPC collective calls.

UPC collective buffer ownership rule: ownership of data involved in collective operations is considered transmitted to the collective upon invocation on each thread, and returned to the user upon completion. The violation of this rule can have unpredictable consequences with respect to completion and results of the collective calls.

The enforcement of the buffer ownership rule is typically left to users. In MPI, and without non-blocking collectives, enforcement is relatively trivial because control is transferred in sync with buffer ownership, leaving no opportunity for the user to execute any offending code.

Non-blocking collectives permit users to violate the ownership rule in multiple ways - by starting a second collective or running other code that accesses memory regions overlapping with buffers already given to the non-blocking collective.

One-sided remote access to buffers in shared memory also give third-party threads (not part of the team executing the collective) the ability to violate the buffer ownership rule.

UPC collective synchronization flags affect the definition of the start and completion of collectives. `IN_ALLSYNC` delays the transfer of buffer from the user to the collective to the point when all participants have started the collective; `OUT_ALLSYNC` delays completion of the collective on any thread to the point where all other threads are also finished. We discuss sync flags in detail in Section 3.1.2.

2.5 Ordering and synchronization of collective operations

Our proposed UPC collectives API supports both blocking and non-blocking collectives. We propose to allow multiple non-blocking collectives, even of the same kind, to be in progress on the same team at the same time. This can lead to non-trivial problems regarding the completion and ordering of collectives, which we must now examine.

The UPC specification introduces six synchronization flags to control data access by collectives. In our experience only two of those six flags are really necessary: `UPC_IN_ALLSYNC` and `UPC_OUT_ALLSYNC`. We describe the proposed syntax in Section 3.1.2; in this section we examine the effect of synchronization flags on collective ordering.

2.5.1 Ordering of blocking collective operations

A blocking collective operation is completed before returning from the function call. Thus any one UPC thread cannot have two blocking collectives in-flight at the same time; for a single thread, program order defines the order in which collectives are executed.

We thus have a simple definition for total order for collectives executing on one thread. The situation is less simple, however, when multiple partially overlapping teams of threads are involved. We define ordering of collectives by means of the directed graph $G(V, E)$ as follows:

- The vertices V of the graph are all collective operations executed in a program.
- A directed edge $v_1 \rightarrow v_2 \in E$ is defined for any pair of collectives where v_1 precedes v_2 on any thread in the program.

Rule for correct ordering for collectives: With the graph defined as above, a UPC program involving collectives is correct only if the directed graph $G(V, E)$ described above has no cycles. The (acyclic) graph also defines a partial order on the execution of collectives in the program. On the other hand if the graph contains any cycles, the corresponding program is faulty and will deadlock.

Synchronization rules for blocking UPC collectives: UPC collectives' synchronization properties are guided by synchronization flags. Execution of a UPC collective on any one thread may be synchronized by activity on other threads, especially when the `ALLSYNC` flags are set.

Discussion: Most MPI collective library implementations have synchronizing properties beyond what is necessary for correctly implementing a particular collective. For example, in an MPI broadcast it is not necessary to synchronize two participating non-root tasks with each other, only with the root; however, most MPI broadcast implementations fully synchronize every pair of participating tasks. The MPI standard warns users not to rely on this property, since it is not intrinsic to the collectives themselves, only a property of the implementation. We make the same stipulation for UPC collectives, subject to synchronization flags.

Advice to users: do not rely on UPC collective calls being synchronizing unless the appropriate sync flags are set.

Advice to implementors: blocking collective implementations are allowed to synchronize the completion of any collective on every executing thread. Correct implementation of `ALLSYNC` flags requires this, but implementors are free to synchronize threads even when `ALLSYNC` flags are not specified by the user.

2.5.2 Ordering of non-blocking collective operations

Each non-blocking collective operation involves two function calls: an initiation call and a completion wait call. A non-blocking collective operation is in-flight when the initiation call is invoked and is complete once the completion wait call returns.

Ordering non-blocking collectives: We order non-blocking collectives by their initiation calls. This approach is consistent with the MPI3 specification and allows construction of the same graph described in Section 2.5.1.

Completion calls are then allowed to be in any order (although they obviously have to follow the initiation calls on the same thread).

Advice to users: A non-blocking collective operation is guaranteed to be complete upon return from `upccoll_wait` or upon `upccoll_test` returning the value 1. However, the collective can complete any time between the initiation call and the completion call. The completion order of multiple in-flight collective operations varies across implementations and different executions. Users should not assume completion orders of collectives beyond the semantics defined by their respective definitions.

Synchronization rules for non-blocking collectives: We clarify this by reusing the collective ordering graph defined in Section 2.5.1. Let us consider the collective start calls $S_i, i \in 0..P-1$ and completion calls $W_i, i \in 0..P-1$ for a particular collective invocation on P threads. Both S and W are vertices in the collective dependence graph $G(V, E)$.

1. We allow edges from any S_i to any W_j vertices (only from S to W, obviously). On any one thread i the initiation call has to precede the completion call; the completion call is allowed to block waiting for the initiation call on potentially all threads.

2. We disallow edges between any two S_i vertices. In vernacular, collective initiation calls should never block waiting initiation of the same collective on other threads. This is a common-sense requirement to allow multiple non-blocking collectives to be launched simultaneously.

   Advice to implementors: The initiation call of any non-blocking collective should never block waiting on any pending activity on some other thread, not even when `IN_ALLSYNC` is specified in the call. `IN_ALLSYNC` simply means that the execution of the collective cannot begin until every thread has started it; it does not constrain the return of the initiation call.

3. We disallow edges between any two W_i vertices. Collective completion calls do not block waiting for completion calls on other threads: they only block waiting for initiation calls on other threads. This results from the common-sense definition of barrier synchronization - the strongest synchronization of all - which states: "a thread is free to *leave* a barrier when every other thread has *entered* the barrier".

   Advice to implementors: The completion call of any non-blocking collective should never block waiting for the corresponding completion call on any other thread, not even when `OUT_ALLSYNC` is specified in the call. `OUT_ALLSYNC` simply means that the collective is complete when the first completion call returns, but this is not subject to completion calls having been invoked on any other threads.

Discussion: First, we should note that the subgraph corresponding to a single non-blocking collective invocation is bipartite, as only edges between S and W are allowed.

The strongest synchronization allowed corresponds to an edge between every pair of W_i and S_j edges (a complete bipartite graph). This synchronization corresponds to a non-blocking barrier. As in the case of blocking collectives, this type of synchronization is only required by `ALLSYNC` flags, therefore users should not rely on it unless they specified `ALLSYNC`.

Advice to users: Do not rely on non-blocking collectives fully synchronizing every thread, unless the `ALLSYNC` flags are specified.

Advice to users: The example code in Figure 1 should not deadlock when executed on multiple threads, even when the collective `ALLSYNC` flags are specified.

2.6 Issues not addressed by our approach

Our present effort to better integrate collective communication into UPC is not complete. We prioritized our approach based on the immediate usefulness of proposed features, and stayed away from topics where discussion in the community is too widely divergent to give us hope of agreement.

- We have not considered MPI-like persistent collectives. We tried very hard to limit the amount of syntax bloat we are adding to the library, and persistent collective communication simply did not make it to the list of included features.

- We have not provided any team management functions other than split. We are well aware of the comparative richness of e.g. MPI in creating new teams by e.g. enumeration. We consider `split` by far the most useful of all team creation primitives, and we are open to further expansion of this functionality in the future.

- We have not considered the effect of teams on UPC index calculation and operations like `upc_global_alloc()`. As we mentioned at the beginning of this Section, our approach does not allow us to change existing language semantics. Even adding a new form of team-based `global_alloc` would have invited the question of how to enumerate (index) the new allocated data structure; this would have been impossible without wide-ranging changes in the semantics of the language.

- We have not considered user defined types such as provided by type constructors in the MPI standard.

- We have not considered value-based collectives, that is UPC collectives on private data. All collectives presented in this paper operate on shared data.

- We have not considered the oft-discussed and published, but never agreed-on concept of one-sided collectives.

/* EX1: lock across start */	/* EX2: lock across wait */

`upccoll_handle_t h;`
`upc_lock_t       lock;`
`    ...`
`upc_lock         (lock);`
`collective_start (... h ...);`
`upc_unlock       (lock);`
`upccoll_wait     (h);`
	`upccoll_handle_t h;`
	`upc_lock_t       lock;`
	`    ...`
	`collective_start (... h ...);`
	`upc_lock         (lock);`
	`upccoll_wait     (h);`
	`upc_unlock       (lock);`

Figure 1: Code sequence EX1 executes a collective start in an atomic section, and will only execute correctly if *collective starts* are non-blocking. Sequence EX2 executes collective completion in an atomic section, and will only complete without deadlock if *collective completions* are not synchronized to each other. Together the two examples enforce the bipartite nature of the dependency graph between collective starts and completions.

3. THE UPC COLLECTIVES 2.0 API

In this section we describe the UPC Collectives API, version 2.0. The API consists of constants, type definitions and function prototypes.

3.1 Type definitions

The UPC Collectives 2.0 API extends types already present in the current UPC specification, but also introduces new types for new concepts.

3.1.1 Collective handles

Synopsis

```
typedef void* upccoll_handle_t;

#define UPCCOLL_INVALID_HANDLE 0
```

Description

1. We define the opaque type `upccoll_handle_t` to represent ongoing collective operations. Handles are returned by non-blocking collective calls, and are also used to test and wait for collection completion (described in Section 3.3).

2. The value 0 is not valid for an ongoing collective.

3.1.2 Collective flag constants

Synopsis

```
typedef int upccoll_flag_t;

#define UPC_IN_NOSYNC   1 /* obsolete */
#define UPC_IN_MYSYNC   2 /* default */
#define UPC_IN_ALLSYNC  4
#define UPC_OUT_NOSYNC  8 /* obsolete */
#define UPC_OUT_MYSYNC  16 /* default */
#define UPC_OUT_ALLSYNC 32
#define UPC_ASYNC_FENCE 64
```

Description

1. This section describes a set of flags that can be specified in the invocation of every collective. The flags affect buffer ownership at the beginning and at the end of a collective, as well as control over the collective function's termination.

2. While all flags defined by the current UPC specification are available in the collectives library, we consider all but three of the flags obsolete and discourage their use. In particular, we consider `UPC_IN_NOSYNC` and `UPC_OUT_NOSYNC` obsolete and to be avoided. We consider `UPC_IN_MYSYNC` and `UPC_OUT_MYSYNC` to correspond to default behavior and therefore do not encourage their explicit use.

3. Buffer ownership is transferred from the user to the system at the moment when a collective is invoked, and is transferred back to the user when the collective terminates. Unless either of the ALLSYNC flags are specified, buffer ownership is transferred to and from the collective locally, without regard to what other threads are doing. For example, late completion of a collective on a remote thread does not prohibit the local thread (where the collective is already complete) from accessing the results.

4. `UPC_IN_ALLSYNC` and `UPC_OUT_ALLSYNC` change buffer ownership transfer. With `IN_ALLSYNC` buffer ownership is not transferred to the collective until *every* participant has invoked the collective. With `OUT_ALLSYNC` no collective is allowed to terminate until all threads are ready to relinquish ownership of buffers.

5. `UPC_ASYNC_FENCE` is the flag that marks a non-blocking collective that will complete at the next call to `upccoll_fence()` (Section 3.3).

6. *Advice to users:* Beware of improper access to shared memory in collectives. There is no mechanism in this specification, or in UPC, to prevent third-party threads from accessing and polluting data processed by collectives. When a team collective is executed on a proper subset of all UPC threads, threads not participating in the collective may inadvertently touch user buffers at the wrong time, producing unexpected results.

7. *Advice to implementors:* The simplest way to implement UPC_IN_ALLSYNC and UPC_OUT_ALLSYNC is by a combined barrier synchronization and memory fence executed at the beginning or the end of the collective, respectively.

8. *Advice to implementors:* Our list includes all flags introduced in the UPC specification. These definitions, less UPC_ASYNC_FENCE, should already be available from the standard include header file upc_collectives.h. Judicious use of conditional macro definitions could solve this problem.

3.1.3 Collective arithmetics

Synopsis

```
typedef enum {
  UPC_ADD,    /* Addition (all types)                         */
  UPC_MULT,   /* Multiplication (all types)                   */
  UPC_AND,    /* Bitwise AND (fixed-point values)             */
  UPC_OR,     /* Bitwise OR  (fixed-point values)             */
  UPC_XOR,    /* Bitwise XOR (fixed-point values)             */
  UPC_LOGAND,/* Logical AND (all types)                       */
  UPC_LOGOR, /* Logical OR  (all types)                       */
  UPC_MIN,    /* Minimum value (all types)                    */
  UPC_MAX,    /* Maximum value (all types)                    */
  UPC_MINLOC,/* Find the minimum value and its location */
  UPC_MAXLOC,/* Find the maximum value and its location */
  UPC_PREDEFINED_OPS /* Number of pre-defined UPC ops   */
}
upccoll_op_t;
```

Description

1. Some of the constants introduced here will be familiar to current users of UPC. They are defined in Section 7.3.2 of the UPC specification.

2. Just like in the UPC specification, the bitwise operators are not defined for floating point operands.

3. New types introduced in this document include UPC_MAXLOC and UPC_MINLOC. These operators work like in MPI.

3.1.4 User-defined collective operations

Synopsis

```
typedef void
upccoll_user_fun  (void             * in,
                   void             * inout,
                   size_t             len,
                   upccoll_dtype_t    dt);

upccoll_return_t
upccoll_op_create (upccoll_user_fun * function,
                   int                commute,
                   upccoll_op_t     * op);

upccoll_return_t
upccoll_op_free   (upccoll_op_t       op);
```

Description

1. The upccoll_op_create function creates a user-defined operation that can be subsequently used in computational collectives including upccoll_reduce, upccoll_allreduce, upccoll_reduce_scatter and upccoll_scan. The new operation bound to the user-defined function is returned in *op.

2. The user-defined operation is always assumed to be associative. If commute is 1, the operation is commutative. If commute is 0, the operation is non-commutative and the order of operands is fixed as ascending UPC thread rank order in the team participating the collective.

3. The upccoll_op_free function frees a user-defined operation.

3.1.5 Built-in collective data types

Synopsis

```
typedef enum {
  UPC_BYTE,           /* sizeof(unsigned char)          */
  UPC_CHAR,           /* sizeof(char)                   */
  UPC_UCHAR,          /* sizeof(unsigned char)          */
  UPC_SHORT,          /* sizeof(short)                  */
  UPC_USHORT,         /* sizeof(unsigned short)         */
  UPC_INT,            /* sizeof(integer)                */
  UPC_UINT,           /* sizeof(unsigned integer)       */
  UPC_LONG,           /* sizeof(long)                   */
  UPC_ULONG,          /* sizeof(unsigned long)          */
  UPC_LONGLONG,       /* sizeof(long long)              */
  UPC_ULONGLONG,      /* sizeof(unsigned long long)     */
  UPC_FLOAT,          /* sizeof(float)                  */
  UPC_DOUBLE,         /* sizeof(double)                 */
  UPC_LONGDOUBLE,     /* sizeof(long double)            */
  UPC_CPLX,           /* 2*sizeof(float)                */
  UPC_DBLCPLX,        /* 2*sizeof(double)               */
  UPC_LONGDBLCPLX,    /* 2*sizeof(long double)          */
  UPC_FLOAT_INT,      /* sizeof(float)+sizeof(int)      */
  UPC_DOUBLE_INT,     /* sizeof(double)+sizeof(int)     */
  UPC_LONG_INT,       /* sizeof(long)+sizeof(int)       */
  UPC_2INT,           /* 2*sizeof(int)                  */
  UPC_SHORT_INT,      /* sizeof(short)+sizeof(int)      */
  UPC_LONG_DOUBLE_INT /* sizeof(long double)+sizeof(int) */
}
upccoll_dtype_t;

upccoll_return_t
upccoll_type_size   (upccoll_dtype_t   dt,
                     size_t          * nbytes);
```

Description

1. In this section we list all pre-defined data types for collective communication. As already mentioned in Section 2.6, this version of the collective specification does not address user defined types.

3.1.6 Error codes

Synopsis

```
typedef enum {
  UPCCOLL_SUCCESS=0,      /* no error                       */
  UPCCOLL_ERROR,          /* generic error                  */
  UPCCOLL_ERROR_TEAM,     /* invalid team handle            */
  UPCCOLL_ERROR_SIZE,     /* invalid team size              */
  UPCCOLL_ERROR_RANK,     /* invalid team rank              */
  UPCCOLL_ERROR_HANDLE,   /* invalid collective handle      */
  UPCCOLL_ERROR_SENDBUF,  /* invalid send buffer (e.g. NULL) */
  UPCCOLL_ERROR_RECVBUF,  /* invalid recv buffer (e.g. NULL) */
  UPCCOLL_ERROR_COUNT,    /* invalid data count (e.g. 0)    */
  UPCCOLL_ERROR_DATATYPE,/* invalid data type specified     */
  UPCCOLL_ERROR_UPC_OP,   /* invalid UPC operation specified */
  UPCCOLL_ERROR_FLAGS,    /* invalid combination of op. flags*/
  UPCCOLL_ERROR_ROOT,     /* invalid root in e.g. broadcast */
  UPCCOLL_ERROR_SENDTYPE,/* invalid send data type          */
  UPCCOLL_ERROR_RECVTYPE,/* invalid receive data type       */
  UPCCOLL_ERROR_SENDCNTS,/* invalid send data count         */
  UPCCOLL_ERROR_RECVCNTS,/* invalid receive data count      */
  UPCCOLL_ERROR_SDISPLS, /* invalid send displacements      */
  UPCCOLL_ERROR_RDISPLS, /* invalid receive displacements   */
  UPCCOLL_ERROR_MALLOC,  /* failure in malloc()             */
  UPCCOLL_ERROR_UNINITIALIZED /* un-initializeded parameter */
}
upccoll_return_t;
```

Description

1. Error codes are returned by every function in our library. We expect a 0 return code to signify success and anything non-zero to signal a failure of some sort.

2. *Advice to implementors:* We supply these error codes as suggestions. Library implementors are not obliged to define or implement anything beyond `UPCCOLL_SUCCESS` and `UPCCOLL_ERROR`.

3.2 Initialization and termination functions

Synopsis

```
upccoll_return_t upccoll_initialize (int *argc, char ***argv);
upccoll_return_t upccoll_finalize   ();
```

Description

1. The initialization function works in a manner similar to `MPI_Init` in the MPI library. No UPC Collectives 2.0 functions should be called before initialization.

2. The initializer is allowed to change the contents of the `argv` array presented to it, removing options it interprets as addressed to the UPC Collectives 2.0 library.

3. The finalize function has barrier semantics. All UPC Collectives 2.0 operations are considered terminated upon return from this call. No UPC Collectives 2.0 functions should be invoked after the return of the finalize function.

3.3 Completion management for non-blocking collectives

Synopsis

```
int   upccoll_test  (upccoll_handle_t h);
void  upccoll_wait  (upccoll_handle_t h);
void  upccoll_fence (void);
```

Description

1. As described in Section 2.3, there are three ways to ensure completion of a UPC Collectives 2.0 function call.

2. If the `UPC_ASYNC_FENCE` flag (see Section 3.1.2) is specified in a UPC Collectives 2.0 call then the call is considered to be non-blocking and will return immediately. Waiting for the call is not necessary; it will complete upon the next invocation of `upccoll_fence` or `upccoll_finalize`.

3. Existing UPC synchronization functions like `upc_barrier` and `upc_fence` have no effect on the completion of `UPC_ASYNC_FENCE` calls. This is in line with our stated principle that we are not altering the semantics of the existing UPC functions.

4. All UPC Collectives 2.0 calls have an out-parameter called `handle`. If `UPC_ASYNC_FENCE` is not set and the pointer supplied to hold the handle is NULL, then the collective call is considered blocking and will be complete upon return.

5. Finally, if `UPC_ASYNC_FENCE` is not set and the user supplies a valid pointer to hold `handle`, the collective call is considered non-blocking and will return a valid handle in that pointer. This operation will not complete until the handle is waited on with `upccoll_wait`.

6. `upccoll_wait` blocks until the operation represented by a handle is complete. Non-blocking operations with valid handles **must** be waited on by `upccoll_wait`.

7. `upccoll_test` can be used to check whether an operation is complete, in a non-blocking manner. However, `upccoll_test` is not a replacement for `upccoll_wait`. The operation cannot be considered complete until the latter function is called.

8. `upccoll_wait` performs the same service for UPC Collectives 2.0 as `upc_wait` does for the standard UPC barrier. Our library approach prohibits us from modifying existing UPC functionality, hence the new name.

3.4 Team operations

1. A UPC team is an ordered collection of unique UPC threads. A team of size $\mathtt{N} \leq \mathtt{THREADS}$ can be thought of as a one-to-one mapping $team : 0..(\mathtt{N}-1) \rightarrow 0..(\mathtt{THREADS}-1)$

2. Teams are created and destroyed by UPC functions described in this section. These functions are similar to MPI communicator management functions. As described in Section 2.1, team creation and destruction functions have collective semantics. In our API teams are identified by *team handles*, an opaque object defined as follows:

```
typedef void* upccoll_team_t;
```

3. Team handles have local semantics only. That is, team IDs should not be stored in shared variables and used across processors. Doing so will result in unpredictable system behavior.

4. **Advice to implementors:** the API is compatible with team handles being local pointers to team objects. There is no guarantee that team objects have the same addresses across different UPC threads.

5. **Default team:** Every collective function call in our library has a team handle argument. The constant value `UPC_TEAM_ALL` can be used in any of these function calls; `UPC_TEAM_ALL` means all UPC threads will participate in the collective.

6. **Advice to implementors:** `UPC_TEAM_ALL` might be implemented as a constant value or a macro with value `NULL`.

3.4.1 *The* `upccoll_team_rank` *function*

Synopsis

```
upccoll_return_t
upccoll_team_rank      (upccoll_team_t team, int *rank);
```

Description

1. The `upccoll_team_rank` function returns the calling thread's rank in a team, a value between 0 and $N-1$ for a team of size N.

2. Since team handles are always local, it follows that only members of a team can query their rank. Different members of a team cannot have the same rank.

3. The value returned by `upccoll_team_rank` when `UPC_TEAM_ALL` is queried is the same as `MYTHREAD`.

3.4.2 *The* `upccoll_team_size` *function*

Synopsis

```
upccoll_return_t upccoll_team_size (upccoll_team_t team, int *size);
```

Description

1. The `upccoll_team_size` function returns the size of a team (the number of UPC threads in the team). This will always be a value between 1 and `THREADS`, and always be `THREADS` for the default team `UPC_TEAM_ALL`. Every team has to have at least one member. All threads in a team will get back the same value for team size.

3.4.3 *The* `upccoll_team_split` *function*

Synopsis

```
upccoll_return_t upccoll_team_split (upccoll_team_t   team,
                                     int              color,
                                     int              key,
                                     upccoll_team_t * newteam);
```

Description

1. Create a set of mutually disjoint new teams from the parent team. This is a collective function, called by every thread in the team.

2. As many new teams are created as the number of distinct `color` identifiers submitted by all threads. Each thread will belong to the team specified by the `color` argument and will have thread ID `key` in that team.

3. Two participating threads in the call cannot specify the same key and color combination (this leads to undefined behavior).

4. For each newly created team, all keys from 0 to size(team)-1 have to be covered by exactly one participant. Failure to do so will result in dysfunctional teams, i.e. undefined behavior when collectives are called on the broken teams.

5. The team handle returned by the operation is a strictly local object. It should not be copied into a shared object and dereferenced by any thread other than the one it was created for.

3.4.4 *The* `upccoll_team_free` *function*

Synopsis

```
void upccoll_team_free (upccoll_team_t team);
```

Description

1. Free the argument team. The function has collective barrier semantics (that is, has to be invoked by every thread in the team at the same time). After the call the team handle cannot be used anymore for any purpose.

3.5 Collective functions

In this section we describe the proposed API for the collective calls themselves. We first enumerate rules that govern all collective calls.

1. In collectives with a `root` argument, all participating threads have to agree on the identity of the root.

2. **Data buffer lengths:** In all collectives the participating threads have to pairwise agree on the size of exchanged data; failure to agree on will cause upredictable results. As in MPI, the size of a buffer is defined as the size associated with the UPC data type of the buffer multiplied by the element count.

3. **Collective flags** are as described in Section 3.1.2. These flags govern buffer ownership rules and collective termination.

4. **Fence semantics:** Collective calls' fence semantics is determined by the flags used in the call, consistent with Section B.3.2.2 in the UPC Specification V1.2.

5. Start and completion order of overlapping collectives is governed by the rules laid down in Section 2.5.

6. **Overlapping send and receive buffers** can yield unpredictable results. The implementation is not required to allocate additional buffer space for holding intermediate results.

7. **Return codes** are detailed in Section 3.1.6.

3.5.1 *The* `upccoll_barrier` *function*

Synopsis

```
upccoll_return_t upccoll_barrier (upccoll_team_t     team,
                                  upccoll_flag_t     flags,
                                  upccoll_handle_t * handle);
```

Description

1. This is a barrier function on the UPC team called `team`.

2. Buffer ownership flags will be ignored, as there are no buffers to exchange.

3. In the non-blocking form of this operation the barrier is complete on any one thread as soon as *every* thread in the team has entered the barrier.

3.5.2 *The* `upccoll_bcast` *function*

Synopsis

```
upccoll_return_t
upccoll_bcast (shared void      * sendbuf,
               size_t             sendcount,
               upccoll_dtype_t    sendtype,
               shared void      * recvbuf,
               size_t             recvcount,
               upccoll_dtype_t    recvtype,
               int                root,
               upccoll_team_t     team,
               upccoll_flag_t     flags,
               upccoll_handle_t * handle);
```

Description

1. This is a broadcast function. (`sendcount`*$size$(`sendtype`) bytes of the send buffer (`sendbuf`) are copied from the *root* UPC thread to all the threads, including itself.

2. The `sendbuf`, `sendcount` and `sendtype` values are ignored by all but the root thread.

3. Every thread (including the root thread) is required to specify the receive buffer `recvbuf`. Every thread receives $size$(`recvtype`) `* recvcnt` bytes.

3.5.3 *The* `upccoll_scatter` *function*

Synopsis

```
upccoll_return_t
upccoll_scatter (shared void        * sendbuf,
                 size_t               sendcnt,
                 upccoll_dtype_t      sendtype,
                 shared void        * recvbuf,
                 size_t               recvcnt,
                 upccoll_dtype_t      recvtype,
                 int                  root,
                 upccoll_team_t       team,
                 upccoll_flag_t       flags,
                 upccoll_handle_t   * handle);
```

Description

1. This is a scatter function. Different parts of the buffer `sendbuf` specified by the `root` thread are copied to `recvbuf` buffers on each thread.

2. Specifically, for every thread $t \in 0..|team| - 1$ in the participating team, $size$(`sendtype`)`*sendcnt` bytes are copied from offset `t*`$size$(`sendtype`)`*sendcnt` of the send buffer to the receive buffer of the thread.

3. The `sendbuf`, `sendcount` and `sendtype` values are ignored by all but the root thread.

4. Every thread (including the root thread) receives $size$(`recvtype`) `* recvcnt` bytes into the receive buffer `recvbuf`.

3.5.4 *The* `upccoll_scatterv` *function*

Synopsis

```
upccoll_return_t
upccoll_scatterv (shared void       * sendbuf,
                  size_t            * sendcnts,
                  size_t            * sdispls,
                  upccoll_dtype_t     sendtype,
                  shared void       * recvbuf,
                  size_t              recvcnt,
                  upccoll_dtype_t     recvtype,
                  int                 root,
                  upccoll_team_t      team,
                  upccoll_flag_t      flags,
                  upccoll_handle_t  * handle);
```

Description

1. This is a scatter function with variable buffer sizes. Portions of the buffer `sendbuf` specified by the `root` thread are copied to the `recvbuf` specified by all threads.

2. Operational semantics are similar to `upccoll_scatter`, except for the amount of data transferred to each participant. The root thread sends `sendcnts[t]*`*size*`(sendtype)` bytes from address `sendbuf + sdispls[t]*`*size*`(sendtype)` to each thread $\texttt{t} \in 0..|\texttt{team}| - 1$.

3. Every thread `t` receives *size*`(recvtype) * recvcnt` bytes into the receive buffer `recvbuf`. This amount has to correspond to the `sendcnts[t]*`*size*`(sendtype)` bytes sent by the root.

3.5.5 *The* `upccoll_gather` *function*

Synopsis

```
upccoll_return_t
upccoll_gather (shared void       * sendbuf,
                size_t              sendcnt,
                upccoll_dtype_t     sendtype,
                shared void       * recvbuf,
                size_t              recvcnt,
                upccoll_dtype_t     recvtype,
                int                 root,
                upccoll_team_t      team,
                upccoll_flag_t      flags,
                upccoll_handle_t  * handle);
```

Description

1. This is a gather function with fixed buffer sizes. The receive buffer `recvbuf` specified by the `root` thread receives a fixed number of bytes from each of the participating threads in the team (including itself).

2. Data sent by thread $\texttt{t} \in 0..|\texttt{team}| - 1$ is received at location `recvbuf + t*recvcnt*`*size*`(recvtype)` on the root thread.

3. On threads other than the root the arguments `recvbuf`, `recvcnt` and `recvtype` are ignored.

4. The number of bytes sent by every thread in the team is *size*`(sendtype)*sendcnt`. This value must correspond to *size*`(recvtype)*recvcnt` specified by the root thread.

3.5.6 *The* `upccoll_gatherv` *function*

Synopsis

```
upccoll_return_t
upccoll_gatherv (shared void       * sendbuf,
                 size_t              sendcnt,
                 upccoll_dtype_t     sendtype,
                 shared void       * recvbuf,
                 size_t            * recvcnts,
                 size_t            * rdispls,
                 upccoll_dtype_t     recvtype,
                 int                 root,
                 upccoll_team_t      team,
                 upccoll_flag_t      flags,
                 upccoll_handle_t  * handle);
```

Description

1. This is a gather function with variable buffer sizes. The receive buffer `recvbuf` specified by the `root` thread receives variable number of bytes from each of the participating threads (as specified by the `team` argument).

2. Data sent by thread $\texttt{t} \in 0..|\texttt{team}| - 1$ is received at offset `rdispls[t]*`*size*`(recvtype)` in the receive buffer on the root thread.

3. On threads other than the root the arguments `recvbuf`, `recvcnts`, `rdispls` and `recvtype` are ignored.

4. The number of bytes sent by every thread `t` in the team is *size*`(sendtype)*sendcnt`. This value must correspond to *size*`(recvtype)*recvcnt[t]` specified by the root thread.

3.5.7 *The* `upccoll_allgather` *function*

Synopsis

```
upccoll_return_t
upccoll_allgather (shared void       * sendbuf,
                   size_t              sendcnt,
                   upccoll_dtype_t     sendtype,
                   shared void       * recvbuf,
                   size_t              recvcnt,
                   upccoll_dtype_t     recvtype,
                   upccoll_team_t      team,
                   upccoll_flag_t      flags,
                   upccoll_handle_t  * handle);
```

Description

1. This is an *allgather* (or all-broadcast) function with fixed buffer sizes. There is no designated root thread: every participant disseminates their send buffer `sendbuf` to every other participant.

2. Data from thread $\mathtt{t} \in 0..|\mathtt{team}|-1$ arrives to offset `recvcnt*`*size*`(recvtype)` in the receive buffer.

3. At the end of the collective the receive buffers on all threads are identical (hence "all-broadcast").

3.5.8 The `upccoll_allgatherv` *function*

Synopsis

```
upccoll_return_t
upccoll_allgatherv (shared void       * sendbuf,
                    size_t              sendcnt,
                    upccoll_dtype_t     sendtype,
                    shared void       * recvbuf,
                    size_t            * recvcnts,
                    size_t            * displs,
                    upccoll_dtype_t     recvtype,
                    upccoll_team_t      team,
                    upccoll_flag_t      flags,
                    upccoll_handle_t  * handle);
```

Description

1. This is an `allgather` function with variable buffer sizes: the participants in the team broadcast variable amounts of data.

2. `sendcnt*`*size*`(sendtype)` bytes are broadcast by each thread to every other thread.

3. Data from thread $\mathtt{t} \in 0..|\mathtt{team}|-1$ arrives to offset `rdispls[t]*`*size*`(recvtype)` in the receive buffer.

4. The number of bytes received from thread `t` is `recvcnts[t]*`*size*`(recvtype)`; this has to correspond to the number of bytes actually sent by thread `t`.

5. At the end of the collective the receive buffers on all threads are identical.

3.5.9 The `upccoll_alltoall` *function*

Synopsis

```
upccoll_return_t
upccoll_alltoall (shared void       * sendbuf,
                  size_t              sendcnt,
                  upccoll_dtype_t     sendtype,
                  shared void       * recvbuf,
                  size_t              recvcnt,
                  upccoll_dtype_t     recvtype,
                  upccoll_team_t      team,
                  upccoll_flag_t      flags,
                  upccoll_handle_t  * handle);
```

Description

1. This is a personalized communication function with fixed buffer sizes. Every participating thread sends `|team|` buffers, one each to every thread in the team, and in turn receives buffers from the same threads. The buffers are all the same size.

2. Data is sent to every thread $\mathtt{t} \in 0..|\mathtt{team}|-1$ from offset `t*sendcnt*`*size*`(sendtype)` of the send buffer `sendbuf`.

3. Data received from thread `t` is deposited at offset `t*recvcnt*`*size*`(recvtype)` of the receive buffer `recvbuf`.

3.5.10 The `upccoll_alltoallv` *function*

Synopsis

```
upccoll_return_t
upccoll_alltoallv (shared void       * sendbuf,
                   size_t            * sendcnts,
                   size_t            * sdispls,
                   upccoll_dtype_t     sendtype,
                   shared void       * recvbuf,
                   size_t            * recvcnts,
                   size_t            * rdispls,
                   upccoll_dtype_t     recvtype,
                   upccoll_team_t      team,
                   upccoll_flag_t      flags,
                   upccoll_handle_t  * handle);
```

Description

1. This is a personalized communication function with variable buffer sizes. Its behavior is similar to that of `alltoall` but the sizes of the exchanged buffers vary.

2. Data destined for thread $\mathtt{t} \in 0..|\mathtt{team}|-1$ is of size `sendcnts[t]*`*size*`(sendtype)` and is sent from offset `sdispls[t]*`*size*`(sendtype)` of the send buffer `sendbuf`.

3. Data received from thread t is expected to be of size `recvcnts[t]*`*size*`(rectype)` and is deposited at offset `rdispls[t]*`*size*`(recvtype)` of the receive buffer `recvbuf`.

3.5.11 The `upccoll_reduce` *function*

Synopsis

```
upccoll_return_t
upccoll_reduce (shared void       * sendbuf,
                shared void       * recvbuf,
                size_t              count,
                upccoll_dtype_t     dt,
                upccoll_op_t        op,
                int                 root,
                upccoll_team_t      team,
                upccoll_flag_t      flags,
                upccoll_handle_t  * handle);
```

Description

1. This is a reduction function with results accumulated in the receive buffer of the `root` thread ($\{\mathtt{root} \in 0..|\mathtt{team}|-1\}$).

2. Every participant provides `count` data items of size *size*`(dt)` bytes each in the send buffer `sendbuf`. The operation `op` is executed on every data element across all participants.

3. Results are available in the receive buffer **recvbuf** provided by the root thread. **recvbuf** is disregarded by all other threads.

4. All participant threads have to agree on the `count`, `dt` and `op` arguments, or else the outcome of the collective function is undefined.

3.5.12 The `upccoll_allreduce` *function*

Synopsis

```
upccoll_return_t
upccoll_allreduce (shared void      * sendbuf,
                   shared void      * recvbuf,
                   size_t             count,
                   upccoll_dtype_t    dt,
                   upccoll_op_t       op,
                   upccoll_team_t     team,
                   upccoll_flag_t     flags,
                   upccoll_handle_t * handle);
```

Description

1. This is a reduction function with results distributed to every participant.

2. Every participant provides `count` data items of size *size*(`dt`) bytes each in the send buffer `sendbuf`. The operation `op` is executed on every data element across all participants.

3. Results are copied to the receive buffers **recvbuf** of every thread.

4. All participant threads have to agree on the `count`, `dt` and `op` arguments, or else the outcome of the collective function is undefined.

3.5.13 The `upccoll_reduce_scatter` *function*

Synopsis

```
upccoll_return_t
upccoll_reduce_scatter (shared void      * sendbuf,
                        shared void      * recvbuf,
                        size_t           * recvcounts,
                        upccoll_dtype_t    dt,
                        upccoll_op_t       op,
                        upccoll_team_t     team,
                        upccoll_flag_t     flags,
                        upccoll_handle_t * handle);
```

Description

1. This is a reduction function with results distributed across the threads.

2. Every participant provides $\sum_{t=0}^{|team|-1} recvcounts[t]$ data items of size *size*(`dt`) bytes each in the send buffer `sendbuf`. The operation `op` is executed on every data element across all participants.

3. Results are distributed in sequence across the receive buffers **recvbuf** provided by all threads, according to the `recvcounts` argument. Thread $\{\mathtt{t} \in 0..|\mathtt{team}| - 1\}$ receives $\mathtt{recvcounts[t]}*size(\mathtt{dt})$ bytes of the end result.

4. All participant threads have to agree on the `dt`, `op` and tt recvcounts arguments, or else the outcome of the collective function is undefined.

3.5.14 The `upccoll_scan` *function*

Synopsis

```
upccoll_return_t
upccoll_scan (shared void        * sendbuf,
              shared void        * recvbuf,
              size_t               count,
              upccoll_dtype_t      dt,
              upccoll_op_t         op,
              upccoll_team_t       team,
              upccoll_flag_t       flags,
              upccoll_handle_t   * handle);
```

Description

1. This is a parallel scan (partial reduction) operation: for a given operator $\bigoplus$ determined by the value of `op`, and contributions $\mathtt{sendbuf_t[i]}, \mathtt{t} \in 0..|\mathtt{team}| - 1\}, \mathtt{i} \in 0..\mathtt{count} - 1,$ the result on thread l is $\mathtt{result_l[i]} = \bigoplus_{\mathtt{k}=0}^{\mathtt{l}-1} \mathtt{sendbuf_k[i]}, \mathtt{i} \in 0..\mathtt{count} - 1.$

2. Every participant provides `count` data items of size *size*(`dt`) bytes each in the send buffer `sendbuf`. Results are copied to the receive buffers **recvbuf** of every thread.

3. All participant threads have to agree on the `count`, `dt` and `op` arguments, or else the outcome of the collective function is undefined.

4. CONCLUSION AND FUTURE WORK

The proposed UPC collectives API described in this paper is by its very nature somewhat of a compromise. Our purpose is to introduce a vision of collective communication to the UPC community. We do not claim our approach to be comprehensive. We have picked the collective operations we consider most important and useful to actual users and lay a groundwork that is reasonably self-contained and open to future expansion. We have self-consciously abstained from a number of features (Section 2.6).

Our hope is that our work may form the kernel of a broad consensus. However, the most important goal is to move the collectives standard in the right direction, whether by our own efforts or someone else's.

5. REFERENCES

[1] The Berkeley UPC Compiler. http://upc.lbl.gov.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, Inc., 1.0α edition, Sept. 2006.

[3] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *The 20th Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[4] D. Bonachea. Proposal for extending the UPC memory copy library functions and supporting extensions to GASNet, v1.0. Technical Report LBNL-56495, Lawrence Berkeley National Laboratory, October 2004.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[6] Cray Inc. The Chapel Parallel Programming Language Home Page. http://chapel.cray.com/ (Mar. 2011).

[7] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. Number v. 2 in Scientific and engineering computation. MIT Press, 1999.

[8] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.

[9] MPI Forum. Message Passing Interface (MPI) Forum Home Page. http://www.mpi-forum.org/ (Dec. 2009).

[10] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009. available at: http://www.mpi-forum.org (Dec. 2009).

[11] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3 (draft), November 2010. available at: http://meetings.mpi-forum.org/MPI_3.0_main_page.php (Nov. 2010).

[12] R. Nishtala, P. H. Hargrove, D. O. Bonachea, and K. A. Yelick. Scaling communication-intensive applications on bluegene/p using one-sided communication and overlap. In *23rd International Parallel & Distributed Processing Symposium*, 2009. Rome, Italy.

[13] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *ACM Fortran Forum*, 17(2):1 – 31, 1998.

[14] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance analysis of mpi collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[15] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[16] R. Thakur and R. Rabenseifner. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19:49–66, 2005.

[17] UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[18] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen. Productivity and performance using partitioned global address space languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, New York, NY, USA, 2007. ACM.

[19] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13), September-November 1998.