GASP: A Performance Tool Interface for Global Address Space Languages

Adam Leko[1], Dan Bonachea[2], Hung-Hsun Su[1], Bryan Golden[1], Hans Sherburne[1], Alan D. George[1]

[1]Electrical Engineering Dept., University of Florida
[2]Computer Science Dept., University of California at Berkeley

Version 1.4, 11/1/2005

1 Introduction

1.1 Scope

Due to the wide range of compilers and the lack of a standardized performance tool interface, writers of performance tools face many challenges when incorporating support for global address space languages such as Unified Parallel C (UPC), Titanium, and Co-Array Fortran (CAF). This document presents a Global Address Space Performance tool interface (GASP) that is flexible enough to be adapted into current global address space compiler and runtime infrastructures with little effort, while allowing performance analysis tools to gather much information about the performance of global address space programs.

1.2 Organization

Section 2 gives a high-level overview of the GASP interface. As GASP can be used to support many languages, the interface has been broken down into language-independent and language-specific sections. Section 3 presents the language-independent portions of the GASP interface, and the following sections detail the language-specific parts of the interface.

1.3 Definitions

- **Users** – individuals using a parallel language such as UPC
- **Developers** – individuals who write parallel software infrastructure such as UPC, CAF, or Titanium compilers
- **Tools** – performance analysis tools such as Vampir, TAU, or KOJAK
- **Tool developers** – individuals who develop performance analysis tools
- **Tool code** – code or library implementing the tool developer's portion of the GASP interface
- **Thread** – a thread of control in a GAS program, maps directly to UPC's concept of threads or CAF's concept of images

2 GASP overview

The GASP interface controls the interaction between a user's code, a performance tool, and GAS language compiler and/or runtime system. This interaction is event-based and comes in the form of callbacks to the `gasp_event_notify` function at runtime. The callbacks may come from instrumentation code placed directly in an executable, from an instrumented runtime library, or any other method; the interface only requires that `gasp_event_notify` is called at appropriate times in the manner described in the rest of this document.

The GASP interface allows tool developers to support GAS languages on all platforms and languages supporting the interface. The interface is used in the following 3 steps:

1. Users compile their GAS code using compiler wrapper scripts provided by tool developers. Users may specify which analysis they wish the tool to perform on their code through either command-line arguments, environment variables or through other tool-specific methods.
2. The compiler wrapper scripts pass appropriate flags to the compiler indicating which callbacks the tool wishes to receive. During the linking phase, the scripts link in appropriate code from the performance tool that handles the callbacks at runtime. This tool-provided code shall be written in C.
3. When a user runs their program, the tool-provided code receives callbacks at runtime and may perform some action such as storing all events in a trace file or performing basic statistical profiling.

The specifics of each step will be discussed in Section 3. The language-specific interface parts of the GASP interface will be discussed in the following sections.

A GAS implementation may exclude any system-level event defined for each language defined in the language-specific sections of this document if an application cannot be instrumented for that event.

Any action resulting in a violation of this specification shall result in undefined behavior. Tool and language implementors are strongly encouraged not to deviate from these specifications.

3 Language-independent interface

3.1 Instrumentation control

Instrumentation control is accomplished through either compilation arguments or compiler pragmas. Developers may use alternative names for the command-line arguments if the names specified below do not fit the conventions already used by the compiler.

3.1.1 User-visible instrumentation control

If a user wishes to instrument their code for use with a tool using the GASP interface, they shall pass either the `--profile` or `--profile-local` command-line arguments to the compiler wrapper scripts.

The `--profile` argument specifies that the user's code shall be instrumented for all events supported by the GAS language implementation, except for events resulting from access to objects or variables contained in the portion of the global address space local to each thread.

For languages that do not have any concept of local or remote memory accesses, this argument shall have the same semantics as the `--profile-local` argument, which specifies that the user's code shall be instrumented for all events support by the GAS language implementation.

3.1.2 Tool-visible instrumentation control

Compilers supporting the GASP interface shall provide three command-line arguments which may be used by the tool-provided compiler wrapper scripts.

The first two arguments `--profile` and `--profile-local` have the same semantics as the user-visible instrumentation flags specified in Section 3.1.1.

The third argument `--profile-only` takes a single argument `filename` which is a file containing a list of symbolic event names (as defined in the language-specific sections of this document) separated by newlines. The file's contents indicate the events for which the performance tool wishes to receive callbacks. Events in this file may be ignored by the compiler if the events are not supported by the GAS language implementation.

3.1.3 Interaction with instrumentation, measurement, and user events

When code is compiled without either the `--profile` or the `--profile-local` flags, all instrumentation control shall be ignored and all user event callbacks shall be compiled away. Systems may link "dummy" versions of `gasp_control` and `gasp_create_event` (described in Sections 3.3 and 3.4) for applications that have no code compiled with `--profile` or `--profile-local`.

Systems may support compiling parts of an application using one of the `--profile` flags and compiling other parts of an application normally; for systems where this is not possible, this behavior may be prohibited. Applications compiled with `--profile` or `--profile-local` on at least one translation unit shall also pass either the `--profile` or `--profile-local` during the linking phase to the compiler wrapper scripts.

Any language-specific instrumentation control shall not have any effect on user events or on the state of measurement control. As a result, any language-specific instrumentation controls shall not prevent user events from being instrumented during compilation (e.g., `#pragma pupc` shall not change the behavior of the `pupc_create_event` and `pupc_event_start` functions in UPC programs).

3.2 Callback structure

During runtime, an instrumented executable shall call the `gasp_init` C function at the beginning of program execution after the language runtime has finished initialization but before executing the entry point in a user's code (e.g., `main` in UPC). The `gasp_init` function shall have the following signature:

```
typedef enum {
  GASP_LANG_UPC,
  GASP_LANG_TITANIUM,
  GASP_LANG_CAF,
  GASP_LANG_MPI,
  GASP_LANG_SHMEM
} gasp_lang_t;

struct _gasp_context_S;
typedef struct _gasp_context_S *gasp_context_t;

gasp_context_t gasp_init(gasp_lang_t srclanguage,
                         int *argc, char ***argv);
```

The `gasp_init` function and an implementation of the `_gasp_context_S` struct shall be provided by tool developers. A single running instance of an executable may call `gasp_init` one or more times if the executable contains code written in multiple languages (such as a hybrid UPC and CAF program).

The `gasp_init` function returns a pointer to a tool-implemented struct that shall be passed in for all subsequent callbacks to the tool developer's code. This pointer shall only be used with events for the language indicated by the `srclanguage` argument.

Tool code may modify the contents of the `argc` and `argv` pointers to support the processing of command-line arguments.

After the `gasp_init` function has been called by each thread of execution, the tool code shall receive all other callbacks through the two functions whose signatures are shown below:

```
typedef enum {
  GASP_START,
  GASP_END,
  GASP_ATOMIC,
} gasp_evttype_t;

void gasp_event_notify(gasp_context_t context, unsigned int evttag,
                       gasp_evttype_t evttype, const char *filename,
                       int linenum, int colnum, ...);

void gasp_event_notifyVA(gasp_context_t context, unsigned int evttag,
                         gasp_evttype_t evttype, const char *filename,
                         int linenum, int colnum, va_list varargs);
```

Both functions may be used interchangeably; the VA variant is provided as a convenience to developers.

The `gasp_event_notify` shall be written in C, but may make upcalls to code written in the language specified by the `srclanguage` argument passed to the `gasp_init` function on the thread that received the callback. If upcalls are used, the `gasp_event_notify` function shall also be re-entrant. Additionally, code that is used in upcalls shall be compiled using the same environmental specifications as the code in a user's application (e.g., `gasp_event_notify` shall only perform upcalls to UPC code compiled under a static threads environment when used with a UPC program compiled under the static threads environment).

Any data referenced by pointers passed to `gasp_event_notify` shall not be changed by tool code.

For the first argument to `gasp_event_notify`, tool code shall receive the same pointer to a `gasp_context_t` that was returned from the `gasp_init` function. Tool developers may use this struct to store thread-local information for each thread. The `gasp_event_notify` function shall be thread-safe for languages that make use of `pthreads` or other thread libraries.

The `evttag` argument shall specify the event identifier as described in the language-specific sections of this document. The `evttype` argument shall be of type `gasp_evttype_t` and shall indicate whether the event `evttag` is a begin event, end event, or atomic event.

The `filename`, `linenum`, and `colnum` arguments shall indicate the source code line and column number that spawned the event `evttag`. GAS language implementations that do not retain column information during compilation may pass 0 in place of the `colnum` parameter. GAS language implementations that do not retain any source-level information during compilation may pass 0 for the `filename`, `linenum`, and `colnum` parameters. GAS language implementations are strongly encouraged to support these arguments unless this information can be efficiently and accurately obtained through other documented methods.

GAS languages that use instrumented runtime libraries for GASP support may provide dummy implementations for the `gasp_event_notify`, `gasp_event_notifyVA`, `gasp_init` functions and `_gasp_context_S` struct to prevent link errors while linking a user's application that is not being used with any performance tool.

The contents of the `varargs` argument shall be specific to each event identifier and type and will be discussed in the language-specific sections of this document.

3.3 Measurement control

Tool developers shall provide an implementation for the following function:

```
int gasp_control(gasp_context_t context, int on);
```

The gasp_control function takes the context argument in the same manner as the gasp_event_notify function.

When the value 0 is sent for the on parameter, the tool shall not measure any performance data (including both system and user events) until the tool code receives another gasp_control call with a nonzero value for the on parameter.

The gasp_control function shall return the last value for the on parameter the function received, or a nonzero value if gasp_control has not been called.

3.4 User events

Tool developers shall provide an implementation for the following function:

```
unsigned int gasp_create_event(gasp_context_t context,
                               const char *name, const char *desc);
```

The gasp_create_event shall return a tool-generated event identifier.

Compilers shall translate the corresponding language-specific _create_event functions listed in the language-specific sections of this document into corresponding gasp_create_event calls. The semantics of the gasp_create_event function shall be the same as semantics of the corresponding _create_event functions listed in the language-specific sections of this document.

3.5 Header files

Developers shall distribute a gasp.h C header file with their GAS language implementations that contains the following definitions:

- Function prototypes for the gasp_init, gasp_event_notify, gasp_control, and gasp_create_event functions and associated typedefs, enums, and structs.
- A GASP_VERSION macro that shall be defined to an integral date (coded as YYYYMMDD) corresponding to the GASP version supported by this GASP implementation. For implementations that support the version of GASP defined in this document, this macro shall be set to the integral value 20051101.
- Macro definitions that map the symbolic event names listed in the language-specific sections of this document to 32-bit unsigned integers.

The gasp.h file shall be installed in a directory that is included in the GAS compiler's default search path.

4 C interface

4.1 Instrumentation control

Instrumentation for the events defined in this section shall be controlled by using the corresponding instrumentation control mechanisms for UPC code defined in Section 5.1.

4.2 Measurement control

Measurement for the events defined in this section shall be controlled by using the corresponding measurement control mechanisms for UPC code defined in Section 5.2.

4.3 User events

4.3.1 Function events

Table 1 shows system events related to executing user functions.

Symbolic name	**Event type**	`vararg` **arguments**
`GASP_C_FUNC`	Start, End	`const char* funcsig`

Table 1: User function events

These events occur when a user function starts and finishes executing. The `funcsig` argument specifies the character string representing the full signature of the function that is executing.

4.4 System events

4.4.1 Memory allocation events

Table 2 shows system events related to the standard memory allocation functions.

Symbolic name	**Event type**	`vararg` **arguments**
`GASP_C_MALLOC`	Start	`size_t nbytes`
`GASP_C_MALLOC`	End	`size_t nbytes, void* returnptr`
`GASP_C_REALLOC`	Start	`void* ptr, size_t size`
`GASP_C_REALLOC`	End	`void* ptr, size_t size, void* returnptr`
`GASP_C_FREE`	Start, End	`void* ptr`

Table 2: Memory allocation events

The `GASP_C_MALLOC`, `GASP_C_REALLOC`, and `GASP_C_FREE` stem directly from the standard C definitions of `malloc`, `realloc`, and `free`.

4.5 Header files

Supported C system events shall be handled in the same method as UPC events, which are described in Section 5.5.

5 UPC interface

5.1 Instrumentation control

Users may insert `#pragma pupc on` or `#pragma pupc off` in their code to instruct the compiler to avoid instrumenting lexically-scoped regions of a user's UPC code. These pragmas may be ignored by the compiler if the compiler cannot control instrumentation for arbitrary regions of code.

When a `--profile` or `--profile-local` argument is given to a compiler or compiler wrapper script, the `#pragma pupc` shall default to `on`.

5.2 Measurement control

At runtime, users may call the following functions below to control the measurement of performance data:

```
int pupc_control(int on);
```

The `pupc_control` function shall behave in the same manner as the `gasp_control` function defined in Section 3.3.

5.3 User events

```
unsigned int pupc_create_event(const char *name, const char *desc);
void pupc_event_start(unsigned int evttag, ...);
void pupc_event_end(unsigned int evttag, ...);
void pupc_event_atomic(unsigned int evttag, ...);
```

The `pupc_create_event` function shall be translated at compile time into a corresponding `gasp_create_event` call defined in Section 3.4. The `name` argument shall be used to associate a user-specified name with the event, and the `desc` argument may contain either `NULL` or a `printf`-style format string.

The event identifier returned by `pupc_create_event` shall be in the range from `GASP_UPC_USEREVT_START` to `GASP_UPC_USEREVT_END`, inclusive. The `GASP_UPC_USEREVT` macros shall be provided in the `gasp_upc.h` header file described in Section 5.5.

The `pupc_event_start`, `pupc_event_end`, and `pupc_event_atomic` functions may be called by a user's UPC program during runtime. The `evttag` argument shall be any value returned by the `pupc_create_event` function. Users may pass in any list of values for the `...` arguments as long as the types passed in match the `printf`-style format string used in the corresponding `pupc_create_event`. A performance tool may use these values to display performance information alongside application-specific data captured during runtime to a user. A compiler shall translate the `pupc_event_start`, `pupc_event_end`, and `pupc_event_atomic` function calls into corresponding `gasp_event_notify` function calls during compilation.

When a compiler does not receive any `--profile` or `--profile-local` arguments, the `pupc_event` function calls shall be excluded from the executable or linked against dummy implementations of these calls. A user's program shall not depend on any side effects that occur from executing the `pupc_event` functions.

Users shall not pass in UPC variables as arguments to the `pupc_event` functions.

5.4 System events

For the event arguments below, the UPC-specific types `upc_flag_t` and `upc_op_t` shall be converted to C `int`s.

Pointers to shared data shall be passed with an extra level of indirection, and may only be dereferenced through UPC upcalls. UPC implementations shall provide two opaque types, `gasp_upc_PTS_t` and `gasp_upc_lock_t`, which shall represent pointer-to-shared and `upc_lock_t`, respectively. These opaque types shall be `typedef`'ed to `void` to prevent C code from attempting to dereference them without using a cast in a UPC upcall.

5.4.1 Exit events

Table 3 shows system events related to the end of a program's execution.

Symbolic name	**Event type**	`vararg` **arguments**
`GASP_UPC_COLLECTIVE_EXIT`	Start, End	`int status`
`GASP_UPC_NONCOLLECTIVE_EXIT`	Atomic	`int status`

Table 3: Exit events

The `GASP_UPC_COLLECTIVE_EXIT` events shall occur at the end of a program's execution on each thread when a collective exit occurs. These events correspond to the execution of the final implicit barrier for UPC programs.

The `GASP_UPC_NONCOLLECTIVE_EXIT` event shall occur at the end of a program's execution on a single thread when a non-collective exit occurs.

5.4.2 Synchronization events

Table 4 shows events related to synchronization constructs.

Symbolic name	**Event type**	`vararg` **arguments**
`GASP_UPC_NOTIFY`	Start, End	`int named, int expr`
`GASP_UPC_WAIT`	Start, End	`int named, int expr`
`GASP_UPC_BARRIER`	Start, End	`int named, int expr`
`GASP_UPC_FENCE`	Start, End	(none)

Table 4: Synchronization events

These events shall occur before and after execution of the notify, wait, barrier, and fence synchronization statements. The `named` argument to the notify, wait, and barrier start events shall be nonzero if the user has provided an integer expression for the corresponding notify, wait, and barrier statements. In this case, the `expr` variable shall be set to the result of evaluating that integer expression. If the user has not provided an integer expression for the corresponding notify, wait, or barrier statements, the `named` argument shall be zero and the value of `expr` shall be undefined.

5.4.3 Work-sharing events

Table 5 shows events related to work-sharing constructs.

Symbolic name	Event type	vararg arguments
GASP_UPC_FORALL	Start, End	(none)

Table 5: Work-sharing events

These events shall occur on each thread before and after `upc_forall` constructs are executed.

5.4.4 Library-related events

Table 6 shows events related to library functions.

Symbolic name	Event type	vararg arguments
GASP_UPC_GLOBAL_ALLOC	Start	`size_t nblocks, size_t nbytes`
GASP_UPC_GLOBAL_ALLOC	End	`size_t nblocks, size_t nbytes, gasp_upc_PTS_t* newshrd_ptr`
GASP_UPC_ALL_ALLOC	Start	`size_t nblocks, size_t nbytes`
GASP_UPC_ALL_ALLOC	End	`size_t nblocks, size_t nbytes, gasp_upc_PTS_t* newshrd_ptr`
GASP_UPC_ALLOC	Start	`size_t nbytes`
GASP_UPC_ALLOC	End	`size_t nbytes, gasp_upc_PTS_t* newshrd_ptr`
GASP_UPC_FREE	Start, End	`gasp_upc_PTS_t* shrd_ptr`
GASP_UPC_GLOBAL_LOCK_ALLOC	Start	(none)
GASP_UPC_GLOBAL_LOCK_ALLOC	End	`gasp_upc_lock_t* lck`
GASP_UPC_ALL_LOCK_ALLOC	Start	(none)
GASP_UPC_ALL_LOCK_ALLOC	End	`gasp_upc_lock_t* lck`
GASP_UPC_LOCK_FREE	Start, End	`gasp_upc_lock_t* lck`
GASP_UPC_LOCK	Start, End	`gasp_upc_lock_t* lck`
GASP_UPC_LOCK_ATTEMPT	Start	`gasp_upc_lock_t* lck`
GASP_UPC_LOCK_ATTEMPT	End	`gasp_upc_lock_t* lck, int result`
GASP_UPC_UNLOCK	Start, End	`gasp_upc_lock_t* lck`
GASP_UPC_MEMCPY	Start, End	`gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, size_t n`
GASP_UPC_MEMGET	Start, End	`void* dst, gasp_upc_PTS_t* src, size_t n`
GASP_UPC_MEMPUT	Start, End	`gasp_upc_PTS_t* dst, void* src, size_t n`
GASP_UPC_MEMSET	Start, End	`gasp_upc_PTS_t* dst, int c, size_t n`

Table 6: Library-related events

These events stem directly from the UPC library functions defined in the UPC specification. The `vararg` arguments for each event callback mirror those defined in the UPC language specification.

5.4.5 Blocking shared variable access events

Table 7 shows events related to blocking shared variable accesses.

Symbolic name	Event type	`vararg` arguments

`GASP_UPC_GET`	Start, End	`int is_relaxed,`
		`void* dst,`
		`gasp_upc_PTS_t* src,`
		`size_t n`

`GASP_UPC_PUT`	Start, End	`int is_relaxed,`
		`gasp_upc_PTS_t* dst,`
		`void* src,`
		`size_t n`

Table 7: Blocking shared variable access events

These events shall occur whenever shared variables are assigned to or read from using the direct syntax (not using the `upc.h` library functions). The arguments to these events mimic those of the `upc_memget` and `upc_memput` event callback arguments, but differ from the ones presented in the previous section because they only arise from accessing shared variables directly. If the memory access occurs under the relaxed memory model, the `is_relaxed` parameter shall be nonzero; otherwise the `is_relaxed` parameter shall be zero.

5.4.6 Nonblocking shared variable access events

Table 8 shows events related to direct shared variable accesses implemented through nonblocking communication.

Symbolic name	Event type	`vararg` arguments

`GASP_UPC_NB_GET_INIT`	Start	`int is_relaxed,`
		`void* dst,`
		`gasp_upc_PTS_t* src,`
		`size_t n`

`GASP_UPC_NB_GET_INIT`	End	`int is_relaxed,`
		`void* dst,`
		`gasp_upc_PTS_t* src,`
		`size_t n,`
		`gasp_upc_nb_handle_t handle`

`GASP_UPC_NB_GET_DATA`	Start, End	`gasp_upc_nb_handle_t handle`

`GASP_UPC_NB_PUT_INIT`	Start	`int is_relaxed,`
		`gasp_upc_PTS_t* dst,`
		`void* src,`
		`size_t n`

`GASP_UPC_NB_PUT_INIT`	End	`int is_relaxed,`
		`gasp_upc_PTS_t* dst,`
		`void* src,`
		`size_t n,`
		`gasp_upc_nb_handle_t handle`

Continued on next page

Symbolic name	Event type	vararg arguments
GASP_UPC_NB_PUT_DATA	Start, End	gasp_upc_nb_handle_t handle
GASP_UPC_NB_SYNC	Start, End	gasp_upc_nb_handle_t handle

Table 8: Nonblocking shared variable access events

These nonblocking direct shared variable access events are similar to the regular direct shared variable access events in Section 5.4.5. The INIT events shall correspond to the nonblocking communication initiation, the DATA events shall correspond to when the data starts to arrive and completely arrives on the destination node (these events may be excluded for most implementations that use hardware-supported DMA), and the GASP_UPC_NB_SYNC function shall correspond to the final synchronization call that blocks until the corresponding data of the nonblocking operation is no longer in flight.

The gasp_upc_nb_handle_t shall be an opaque type to tool developers defined by a UPC implementation. Several outstanding nonblocking get or put operations may be attached to a single gasp_upc_nb_handle_t instance. When a sync callback is received, the tool code shall assume all get and put operations for the corresponding handle in the sync callback have been retired.

The implementation may pass the handle GASP_NB_TRIVIAL to GASP_UPC_NB_{PUT,GET}_INIT to indicate the operation was completed synchronously in the initiation interval. The tool should ignore any DATA or SYNC event callbacks with the handle GASP_NB_TRIVIAL.

5.4.7 Shared variable cache events

Table 9 shows events related to shared variable cache events.

Symbolic name	Event type	vararg arguments
GASP_UPC_CACHE_MISS	Atomic	size_t n, size_t n_lines
GASP_UPC_CACHE_HIT	Atomic	size_t n
GASP_UPC_CACHE_INVALIDATE	Atomic	size_t n_dirty

Table 9: Shared variable cache events

The GASP_UPC_CACHE events may be sent for UPC runtime systems containing a software cache after a corresponding get or put start event but before a corresponding get or put end event (including nonblocking communication events). UPC runtimes using write-through cache systems may send GASP_UPC_CACHE_MISS events for each corresponding put event.

The size_t n argument for the MISS and HIT events shall indicate the amount of data read from the cache line for the particular cache hit or cache miss.

The n_lines argument of the GASP_UPC_CACHE_MISS event shall indicate the number of bytes brought into the cache as a result of the miss (in most cases, the line size of the cache).

The n_dirty argument of the GASP_UPC_CACHE_INVALIDATE shall indicate the number of dirty cache lines that were written back to shared memory due to a cache line invalidation.

5.4.8 Collective communication events

Table 10 shows events related to collective communication.

Symbolic name	Event type	vararg arguments
GASP_UPC_ALL_BROADCAST	Start, End	gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, size_t nbytes, **int** upc_flags
GASP_UPC_ALL_SCATTER	Start, End	gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, size_t nbytes, **int** upc_flags
GASP_UPC_ALL_GATHER	Start, End	gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, size_t nbytes, **int** upc_flags
GASP_UPC_ALL_GATHER_ALL	Start, End	gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, size_t nbytes, **int** upc_flags
GASP_UPC_ALL_EXCHANGE	Start, End	gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, size_t nbytes, **int** upc_flags
GASP_UPC_ALL_PERMUTE	Start, End	gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, gasp_upc_PTS_t* perm, size_t nbytes, **int** upc_flags
GASP_UPC_ALL_REDUCE	Start, End	gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, **int** upc_op, size_t nelems, size_t blk_size, **void*** func, **int** upc_flags, gasp_upc_reduction_t type
GASP_UPC_ALL_PREFIX_REDUCE	Start, End	gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, **int** upc_op, size_t nelems, size_t blk_size, **void*** func, **int** upc_flags, gasp_upc_reduction_t type

Table 10: Collective communication events

The events in Table 10 stem directly from the UPC collective library functions defined in the UPC specification. The `vararg` arguments for each event callback mirror those defined in the UPC language specification.

For the reduction functions, the `gasp_upc_reduction_t` enum shall be provided by a UPC implementation and shall be defined as follows:

```
typedef enum {
  GASP_UPC_REDUCTION_C,
  GASP_UPC_REDUCTION_UC,
```

```
  GASP_UPC_REDUCTION_S,
  GASP_UPC_REDUCTION_US,
  GASP_UPC_REDUCTION_I,
  GASP_UPC_REDUCTION_UI,
  GASP_UPC_REDUCTION_L,
  GASP_UPC_REDUCTION_UL,
  GASP_UPC_REDUCTION_F,
  GASP_UPC_REDUCTION_D,
  GASP_UPC_REDUCTION_LD
} gasp_upc_reduction_t;
```

where the suffix to `GASP_UPC_REDUCTION` denotes the same type as specified in the UPC specification.

5.5 Header files

UPC compilers shall distribute a `pupc.h` C header file with their GAS language implementations that contains function prototypes for the functions defined in Sections 5.2 and 5.3. The `pupc.h` file shall be installed in a directory that is included in the UPC compiler's default search path.

All supported system events shall be defined in a `gasp_upc.h` file located in the same directory as the `gasp.h` file. System events not supported by an implementation shall not be included in the `gasp_upc.h` file. The `gasp_upc.h` header file may include definitions for implementation-specific events, along with brief documentation embedded in source code comments.

Compilers shall define a compiler-specific integral `GASP_UPC_VERSION` version number in `gasp_upc.h` that may be incremented when new implementation-specific events are added. Compiler developers are encouraged to use the `GASP_X_Y` naming convention for all implementation-specific events, where `X` is an abbreviation for their compilation system (such as `BUPC`) and `Y` is a short, descriptive name for each event.

6 Titanium interface

TBD

7 CAF interface

TBD

8 MPI interface

TBD

9 SHMEM interface

TBD