MPPA® Asynchronous Communication API

Kalray S.A.

This document was compiled in *debug* mode.
It is not suitable for release.

32 pages

MPPA® Asynchronous Communication API

Kalray S.A.[1]

[1] info@kalray.eu, Kalray S.A.

Abstract: This document specifies the MPPA® Asynchronous Operations application programming interface (API). This API is designed to support application in the compute clusters that need to access remote memory, whether the DDR memory or the SMEM of other compute clusters. This API is directly available from the Low-Level and POSIX-Level programming environments of the MPPA® platform. A subset of its functionality is also exposed under the standard OpenCL asynchronous copy operations.

Keywords: MPPA, one-sided communication, asynchronous data transfer, API

Contents

1 Introduction

This document presents an API for asynchronous operations based on one-sided communications between the MPPA® compute cluster local memories and I/O cluster DDR memories. The motivation is to apply to the MPPA® platform the well-known principles of one-sided communications libraries of supercomputers, in particular: the Cray SHMEM library [1], the PNNL ARMCI library [4], and the MPI-2 one-sided API subset [2]. The main difference between these and the MPPA® asynchronous operations is that a supercomputer has a *symmetric* architecture, where the compute nodes are identical and the working memory is composed of the union of the compute node local memories. In case of the MPPA® architecture, the compute cluster local memories have a limited capacity so transfers are required between these memories and the external DDR memories.

The MPPA® platform already supports transfers of data between the compute cluster local memories and the external DDR memories, based on a software distributed shared memory (S-DSM) implementation inspired by Treadmarks [3] . The S-DSM leverages the MMU available on each PE core to convert part of the local memory into a last-level cache. The problem is that each such cache line has the size of a MMU pages, typically $4K$ or $8K$ bytes. This implies that only a limited number of such cache lines may fit the compute cluster local memory, and the large size of each line require significant spatial locality before the miss penalty can be amortized. Finally, the temporal behavior of a (software) cache system is difficult to control and this is an issue for time-critical applications.

The MPPA® asynchronous operations library can be used either alone or in combination with the MPPA® platform S-DSM. This API is directly available from the Low-Level and POSIX-Level programming environments of the MPPA® platform. A subset of its functionality is also exposed under the standard OpenCL asynchronous copy operations.

2 Asynchronous Operations API Overview

The MPPA® Asynchronous Operations API is organized around several concepts:

Execution Domains An execution domain is a set of cores sharing local memory, which can be isolated from other domains. On the MPPA® platform, an execution domain corresponds to a compute cluster, or to a partition of an I/O cluster.

Memory Segments Memory that is not directly accessible from the cores of an execution domain is structured into segments, which correspond to whole or part of the local memory of cores located in another execution domain.

PUT/GET Operations These operations initiate an asynchronous write or read between the local memory and a designated memory segment. Two addresses are supplied, a local one and a remote relative to the memory segment. Two families of PUT/GET operations are provided, one where the local data accesses are dense (Section 6) and the other where the local data accesses are sparse (Section 7).

Asynchronous Operations All PUT/GET operations, the remote memory operations (Section 8.1) and most remote synchronization operations (Section 8) are asynchronous, meaning the operation call returns as soon as the local memory can be reused. The last parameter of such call is a pointer to an event structure. If NULL, the operation blocks until completion. If given a pointer, the event structure is initialized and can be later waited or tested for the operation completion.

Operation Completion Operation completion can be local or remote. Local operation completion means that the local object can be reused in case of PUT, and has been updated in case of GET. Remote operation completion means that the effect in the remote segment is visible from a core of the execution domain where this segment is located. Both types of operation completion are converted to event completion on the event structure that has been initialized by the operation. Event completion is then is waited for or tested by calling the corresponding functions (Section 4).

Memory Consistency No ordering can be assumed between memory operations directed to different segments. Inside a remote memory segment operated from the same domain, GET operations are unordered and PUT operations are only ordered between themselves. Both PUT/GET operations in a segment are ordered with respect to a remote synchronization operation (Section 8) directed to that segment. Finally, all remote synchronization operations from the same domain and targeted to the same segment are are ordered.

Remote Synchronization Remote synchronization is achieved by updating a 64-bit location in a remote memory segment. This is done either by calling a fence operation after one or more PUT operations, or by accessing this remote 64-bit location by calling a peek, poke, or atomic operation. A core operating locally on the segment with the 64-bit location can then evaluate if the value of that location passes a condition (e.g. non-zero). The evaluation itself can be blocking if the event pointer supplied is NULL. Else, the evaluation of the condition is deferred to the moment the event completion will be waited for.

Remote Memory Allocation In order to support application programming without developping code for the I/O clusters, functions for memory allocation in a remote memory segment are provided.

3 Asynchronous Operations Context

3.1 Asynchronous Operations Library

3.1.1 mppa_async_init

```
/**
 * Initialize the asynchronous operation library.
 * return 0 on success.
 */
int
mppa_async_init(void);
```

3.1.2 mppa_async_final

```
/**
 * Finalize the asynchronous operation library.
 * return 0 on success.
 */
int
mppa_async_final(void);
```

4 Asynchronous Operation Events

4.1 Event Data Types

4.1.1 mppa_async_event_t

```
/**
 * Asynchronous operation event type.
 */
typedef union {
  MPPA_ASYNC_EVENT_STRUCT;
  long long payload[4];
} mppa_async_event_t;
```

4.2 Event Wait Operations

4.2.1 mppa_async_event_wait

```
/**
 * Wait for an event to occur (blocking function).
 * event The event to wait for.
 * return 0 on success or NULL event.
 */
int
mppa_async_event_wait(mppa_async_event_t *event);
```

4.2.2 mppa_async_event_waitall

```
/**
 * Wait for all events in an array to occur.
 * count Count of elements in events.
 * events Array of events to wait for.
 * return 0 on success.
 */
int
mppa_async_event_waitall(int count, mppa_async_event_t events[]);
```

4.2.3 mppa_async_event_waitany

```
/**
 * Wait for any event in an array to occur.
 * count Count of elements in events.
 * events Array of events to wait for.
 * return 0 on success.
 */
int
mppa_async_event_waitany(int count, mppa_async_event_t events[]);
```

4.3 Event Test Operations

4.3.1 mppa_async_event_test

```
/**
 * Test for an event to occur (non-blocking).
 * event Event to try wait for.
 * return 0 on success or NULL event, -1 if not done.
 */
int
mppa_async_event_test(mppa_async_event_t *event);
```

4.3.2 mppa_async_event_testall

```
/**
 * Test for all events in an array to occur (non-blocking).
 * count Count of elements in events.
 * events Array of events to wait for.
 * return 0 on success, -n if n not done.
 */
int
mppa_async_event_testall(int count, mppa_async_event_t events[]);
```

4.3.3 mppa_async_event_testany

```
/**
 * Test for any event in an array to occur (non-blocking).
 * count Count of elements in events.
 * events Array of events to wait for.
 * return 0 on success.
 */
int
mppa_async_event_testany(int count, mppa_async_event_t events[]);
```

5 Asynchronous Operation Segments

5.1 Segment Data Types

5.1.1 mppa_async_segment_t

```
/*
 * Descriptor of a memory segment.
 */
typedef struct mppa_async_segment mppa_async_segment_t;
```

5.1.2 mppa_async_segment_flag_t

```
/**
 * Flags for the mppa_async_segment_t creation.
 */
typedef enum mppa_async_segment_flag {
  MPPA_ASYNC_SEGMENT_FLAG_QUEUE0 =  0x1, /* Queue on interface 0. */
  MPPA_ASYNC_SEGMENT_FLAG_QUEUE1 =  0x2, /* Queue on interface 1. */
  MPPA_ASYNC_SEGMENT_FLAG_QUEUE2 =  0x4, /* Queue on interface 2. */
  MPPA_ASYNC_SEGMENT_FLAG_QUEUE3 =  0x8, /* Queue on interface 3. */
} mppa_async_segment_flag_t;
```

5.2 Segment Life Cycle

5.2.1 mppa_async_segment_create

```
/**
 * Create a segment on the local memory server.
 * segment Pointer to the segment object in local memory.
 * base Base address of the segment in local memory.
 * size Size in bytes of the segment in local memory.
 * flags Flags of the segment by OR-ing mppa_async_segment_flag_t values.
 * multi If non-zero, collectively create a multicast segment for this number.
 * ident Identifier assigned to the segment.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_segment_create(mppa_async_segment_t *segment,
                          void *base, size_t size,
                          unsigned flags, int multi,
                          unsigned long long ident,
                          mppa_async_event_t *event);
```

5.2.2 mppa_async_segment_clone

```
/**
 * Clone a segment created on a remote memory server.
 * segment Pointer to the segment object in local memory.
 * ident Identifier of the segment to clone.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_segment_clone(mppa_async_segment_t *segment,
                         unsigned long long ident,
                         mppa_async_event_t *event);
```

5.2.3 mppa_async_segment_destroy

```
/**
 * Destroy a segment on the local memory server.
 * segment Pointer to the segment object in local memory.
 * return 0 on success.
 */
int
mppa_async_segment_destroy(mppa_async_segment_t *segment);
```

5.3 Segment Access

5.3.1 mppa_async_default_segment

```
/**
 * Get the default segment of a memory server.
 * server The identifier of the memory server.
 * return Const pointer to the segment, or NULL if not found.
 */
static inline
const mppa_async_segment_t *
mppa_async_default_segment(int server)
{
  return MPPA_ASYNC_DEFAULT_SEGMENT(server);
}
```

6 Dense Asynchronous Transfers

6.1 Contiguous Asynchronous Transfers

6.1.1 mppa_async_get

```
/**
 * Start a contiguous asynchronous transfer from remote memory to local memory.
 * local Local memory pointer where the data will be copied to.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the data will be copied from.
 * bytes Size in bytes the data to transfer.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_get(void *local, const mppa_async_segment_t *segment,
               off64_t offset, size_t bytes,
               mppa_async_event_t *event);
```

6.1.2 mppa_async_put

```
/**
 * Start a contiguous asynchronous transfer from local memory to remote memory.
 * local Local memory pointer where the data will be copied from.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the data will be copied to.
 * bytes Size in bytes the data to transfer.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_put(const void *local, const mppa_async_segment_t *segment,
               off64_t offset, size_t bytes,
               mppa_async_event_t *event);
```

6.2 Spaced Asynchronous Transfers

6.2.1 mppa_async_get_spaced

```
/**
 * Start a spaced asynchronous transfer from remote memory to local memory.
 * local Local memory pointer where the data will be copied to.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the data will be copied from.
 * size Size in bytes of one data element.
 * count Count of elements to transfer (total transfer size is size*count).
 * space Space between remote elements in bytes.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_get_spaced(void *local, const mppa_async_segment_t *segment,
                      off64_t offset, size_t size, int count, size_t space,
                      mppa_async_event_t *event);
#define mppa_async_get_strided(local, segment, offset, size, count, stride, event) \
  mppa_async_get_spaced(local, segment, offset, size, count, stride*size, event)
```

This function is a higher performance implementation of the following code:

```
int
mppa_async_get_spaced(void *local, const mppa_async_segment_t *segment,
                      off64_t offset, size_t size, int count, size_t space,
                      mppa_async_event_t *event)
{
  int status = 0;
  off64_t remote_offset = offset;
  char *local_address = (char *)local;
  for (int i = 0; i < count; i++) {
    status |= mppa_async_get(local_address, segment, remote_offset, size, event);
    remote_offset += space;
    local_address += size;
  }
  return status;
}
```

6.2.2 mppa_async_put_spaced

```
/**
 * Start a spaced asynchronous transfer from local memory to remote memory.
 * local Local memory pointer where the data will be copied from.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the data will be copied to.
 * size Size in bytes of one data element.
 * count Count of elements to transfer (total transfer size is size*count).
 * space Space between remote elements in bytes.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_put_spaced(const void *local, const mppa_async_segment_t *segment,
                      off64_t offset, size_t size, int count, size_t space,
                      mppa_async_event_t *event);
#define mppa_async_put_strided(local, segment, offset, size, count, stride, event) \
  mppa_async_put_spaced(local, segment, offset, size, count, stride*size, event)
```

This function is a higher performance implementation of the following code:

```
int
mppa_async_put_spaced(const void *local, const mppa_async_segment_t *segment,
                      off64_t offset, size_t size, int count, size_t space,
                      mppa_async_event_t *event)
{
  int status = 0;
  off64_t remote_offset = offset;
  const char *local_address = (const char *)local;
  for (int i = 0; i < count; i++) {
    status |= mppa_async_put(local_address, segment, remote_offset, size, event);
    remote_offset += space;
    local_address += size;
  }
  return status;
}
```

6.3 Vectored Asynchronous Transfers

6.3.1 mppa_async_get_vectored

```
/**
 * Start a vectored asynchronous copy from remote memory to local memory.
 * local Local memory pointer where the data will be copied to.
 * segment Identifies the remote memory segment.
```

```
 * count Count of elements in arrays offsets and sizes.
 * offsets Array of offset values that specifies the transfer.
 * sizes Array of size values that specifies the transfer.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_get_vectored(void *local, const mppa_async_segment_t *segment,
                        int count, off64_t offsets[], size_t sizes[],
                        mppa_async_event_t *event);
```

This function is a higher performance implementation of the following code:

```
int
mppa_async_get_vectored(void *local, const mppa_async_segment_t *segment,
                        int count, off64_t offsets[], size_t sizes[],
                        mppa_async_event_t *event)
{
  int status = 0;
  char *local_address = (char *)local;
  for (int i = 0; i < count; i++) {
    size_t size = sizes[i];
    status |= mppa_async_get(local_address, segment, offsets[i], size, event);
    local_address += size;
  }
  return status;
}
```

6.3.2 mppa_async_put_vectored

```
/**
 * Start a vectored asynchronous copy from local memory to remote memory.
 * local Local memory pointer where the data will be copied from.
 * segment Identifies the remote memory segment.
 * count Count of elements in arrays offsets and sizes.
 * offsets Array of offset values that specifies the transfer.
 * sizes Array of size values that specifies the transfer.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_put_vectored(const void *local, const mppa_async_segment_t *segment,
                        int count, off64_t offsets[], size_t sizes[],
                        mppa_async_event_t *event);
```

This function is a higher performance implementation of the following code:

```
int
mppa_async_put_vectored(const void *local, const mppa_async_segment_t *segment,
                        int count, off64_t offsets[], size_t sizes[],
                        mppa_async_event_t *event)
{
  int status = 0;
  const char *local_address = (const char *)local;
  for (int i = 0; i < count; i++) {
    size_t size = sizes[i];
    status |= mppa_async_put(local_address, segment, offsets[i], size, event);
    local_address += size;
  }
  return status;
}
```

6.4 Indexed Asynchronous Transfers

6.4.1 mppa_async_get_indexed

```
/**
 * Start an indexed asynchronous transfer from remote memory to local memory.
 * local Local memory pointer where the data will be copied to.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the data will be copied from.
 * size Size in bytes of one data element.
 * count Count of elements to transfer (total transfer size is size*count).
 * indices Array of count indices, each yielding an offset by scaling by size.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_get_indexed(void *local, const mppa_async_segment_t *segment,
                       off64_t offset, size_t size, int count, const int indices[],
                       mppa_async_event_t *event);
```

This function is a higher performance implementation of the following code:

```
int
mppa_async_get_indexed(void *local, const mppa_async_segment_t *segment,
                       off64_t offset, size_t size, int count, const int indices[],
                       mppa_async_event_t *event)
{
  int status = 0;
  char *local_address = (char *)local;
  for (int i = 0; i < count; i++) {
    status |= mppa_async_get(local_address, segment, offset+indices[i]*size, size,
        event);
    local_address += size;
  }
  return status;
}
```

6.4.2 mppa_async_put_indexed

```
/**
 * Start an indexed asynchronous transfer from local memory to remote memory.
 * local Local memory pointer where the data will be copied from.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the data will be copied to.
 * size Size in bytes of one data element.
 * count Count of elements to transfer (total transfer size is size*count).
 * indices Array of count indices, each yielding an offset by scaling by size.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_put_indexed(const void *local, const mppa_async_segment_t *segment,
                       off64_t offset, size_t size, int count, const int indices[],
                       mppa_async_event_t *event);
```

This function is a higher performance implementation of the following code:

```
int
mppa_async_put_indexed(const void *local, const mppa_async_segment_t *segment,
                       off64_t offset, size_t size, int count, const int indices[],
                       mppa_async_event_t *event)
{
```

```
  int status = 0;
  const char *local_address = (const char *)local;
  for (int i = 0; i < count; i++) {
    status |= mppa_async_put(local_address, segment, offset+indices[i]*size, size,
        event);
    local_address += size;
  }
  return status;
}
```

6.5 Streamed Asynchronous Transfers

6.5.1 mppa_async_get_streamed

```
/**
 * Start a streamed asynchronous transfer from remote memory to local memory.
 * local Local memory pointer where the data will be copied to.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the data will be copied from.
 * size Size in bytes of one data element.
 * count Count of elements to transfer (total transfer size is size*count).
 * stride Stride between remote elements (1 for consecutive elements).
 * span Number of elements before applying the skip.
 * skip Number of elements skipped when the element counter is multiple of span.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_get_streamed(void *local, const mppa_async_segment_t *segment,
                        off64_t offset, size_t size, int count,
                        int stride, int span, int skip,
                        mppa_async_event_t *event);
```

This function is a higher performance implementation of the following code:

```
int
mppa_async_get_streamed(void *local, const mppa_async_segment_t *segment,
                        off64_t offset, size_t size, int count,
                        int stride, int span, int skip,
                        mppa_async_event_t *event)
{
  int status = 0;
  off64_t remote_offset = offset;
  char *local_address = (char *)local;
  int position = span;
  for (int i = 0; i < count; i++) {
    status |= mppa_async_get(local_address, segment, remote_offset, size, event);
    remote_offset += stride*size;
    local_address += size;
    if (i + 1 == position) {
      remote_offset += skip*size;
      position += span;
    }
  }
  return status;
}
```

6.5.2 mppa_async_put_streamed

```
/**
 * Start a streamed asynchronous transfer from local memory to remote memory.
 * local Local memory pointer where the data will be copied from.
```

```
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the data will be copied to.
 * size Size in bytes of one data element.
 * count Count of elements to transfer (total transfer size is size*count).
 * stride Stride between remote elements (1 for consecutive elements).
 * span Number of elements before applying the skip.
 * skip Number of elements skipped when the element counter is multiple of span.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_put_streamed(const void *local, const mppa_async_segment_t *segment,
                        off64_t offset, size_t size, int count,
                        int stride, int span, int skip,
                        mppa_async_event_t *event);
```

This function is a higher performance implementation of the following code:

```
int
mppa_async_put_streamed(const void *local, const mppa_async_segment_t *segment,
                        off64_t offset, size_t size, int count,
                        int stride, int span, int skip,
                        mppa_async_event_t *event)
{
  int status = 0;
  off64_t remote_offset = offset;
  const char *local_address = (const char *)local;
  int position = span;
  for (int i = 0; i < count; i++) {
    status |= mppa_async_put(local_address, segment, remote_offset, size, event);
    remote_offset += stride*size;
    local_address += size;
    if (i + 1 == position) {
      remote_offset += skip*size;
      position += span;
    }
  }
  return status;
}
```

7 Sparse Asynchronous Transfers

7.1 Spaced Asynchronous Transfers

7.1.1 mppa_async_sget_spaced

```
/**
 * Start a spaced stepped asynchronous transfer from remote memory to local memory.
 * local Local memory pointer where the data will be copied to.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the data will be copied from.
 * size Size in bytes of one data element.
 * count Count of elements to transfer (total transfer size is size*count).
 * skew Skew between local elements in bytes.
 * space Space between remote elements in bytes.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_sget_spaced(void *local, const mppa_async_segment_t *segment,
                       off64_t offset, size_t size, int count,
                       size_t skew, size_t space,
                       mppa_async_event_t *event);
```

This function is a higher performance implementation of the following code:

```
int
mppa_async_sget_spaced(void *local, const mppa_async_segment_t *segment,
                       off64_t offset, size_t size, int count,
                       size_t skew, size_t space,
                       mppa_async_event_t *event)
{
  int status = 0;
  off64_t remote_offset = offset;
  char *local_address = (char *)local;
  for (int i = 0; i < count; i++) {
    status |= mppa_async_get(local_address, segment, remote_offset, size, event);
    remote_offset += space;
    local_address += skew;
  }
  return status;
}
```

7.1.2 mppa_async_sput_spaced

```
/**
 * Start a spaced stepped asynchronous transfer from local memory to remote memory.
 * local Local memory pointer where the data will be copied from.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the data will be copied to.
 * size Size in bytes of one data element.
 * count Count of elements to transfer (total transfer size is size*count).
 * skew Skew between local elements in bytes.
 * space Space between remote elements in bytes.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_sput_spaced(const void *local, const mppa_async_segment_t *segment,
                       off64_t offset, size_t size, int count,
                       size_t skew, size_t space,
```

```
                              mppa_async_event_t *event);
```

This function is a higher performance implementation of the following code:

```
int
mppa_async_sput_spaced(const void *local, const mppa_async_segment_t *segment,
                       off64_t offset, size_t size, int count,
                       size_t skew, size_t space,
                       mppa_async_event_t *event)
{
  int status = 0;
  off64_t remote_offset = offset;
  const char *local_address = (const char *)local;
  for (int i = 0; i < count; i++) {
    status |= mppa_async_put(local_address, segment, remote_offset, size, event);
    remote_offset += space;
    local_address += skew;
  }
  return status;
}
```

7.2 Block 2D Asynchronous Transfers

7.2.1 mppa_async_point2d_t

```
typedef struct {
  int xpos;
  int ypos;
  int xdim;
  int ydim;
} mppa_async_point2d_t;
```

7.2.2 mppa_async_sget_block2d

```
/**
 * Start a 2D asynchronous transfer from the remote submatrix to the local submatrix.
 * local Local memory pointer where the data will be copied to.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the data will be copied from.
 * size Size in bytes of one data element.
 * width Width of the submatrix (x coordinate).
 * height Height of the submatrix (y coordinate).
 * local_point Point in the local submatrix.
 * remote_point Point in the remote submatrix.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_sget_block2d(void *local, const mppa_async_segment_t *segment,
                        off64_t offset, size_t size, int width, int height,
                        const mppa_async_point2d_t *local_point,
                        const mppa_async_point2d_t *remote_point,
                        mppa_async_event_t *event);
```

This function is a higher performance implementation of the following code:

```
int
mppa_async_sget_block2d(void *local, const mppa_async_segment_t *segment,
                        off64_t offset, size_t size, int width, int height,
                        const mppa_async_point2d_t *local_point,
                        const mppa_async_point2d_t *remote_point,
                        mppa_async_event_t *event)
```

```
{
  int status = 0;
  mppa_async_event_t local_event;
  mppa_async_event_t *_event = (event==NULL)? &local_event: event;
  off64_t remote_offset = offset +
                          (remote_point->xpos +
                          (remote_point->ypos*remote_point->xdim))*size;
  char *local_address = (char *)local +
                        (local_point->xpos +
                        (local_point->ypos*local_point->xdim))*size;
  // if extent is bigger than local, in any of dimensions, then return -1
  if ((width  > (local_point->xdim - local_point->xpos)) ||
      (height > (local_point->ydim - local_point->ypos))) {
    status = -1;
  } else {
    status |= mppa_async_sget_spaced(local_address, segment, remote_offset,
      width*size,                /* size */
      height,                    /* count */
      local_point->xdim*size,    /* skew */
      remote_point->xdim*size,   /* space */
      _event);
  }
  if (event==NULL) {
    status = mppa_async_event_wait(_event);
  }
  return status;
}
```

7.2.3 mppa_async_sput_block2d

```
/**
 * Start a 2D asynchronous transfer from the local submatrix to the remote submatrix.
 * local Local memory pointer where the data will be copied from.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the data will be copied to.
 * size Size in bytes of one data element.
 * width Width of the submatrix (x coordinate).
 * height Height of the submatrix (y coordinate).
 * local_point Point in the local submatrix.
 * remote_point Point in the remote submatrix.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_sput_block2d(const void *local, const mppa_async_segment_t *segment,
                        off64_t offset, size_t size, int width, int height,
                        const mppa_async_point2d_t *local_point,
                        const mppa_async_point2d_t *remote_point,
                        mppa_async_event_t *event);
```

This function is a higher performance implementation of the following code:

```
int
mppa_async_sput_block2d(const void *local, const mppa_async_segment_t *segment,
                        off64_t offset, size_t size, int width, int height,
                        const mppa_async_point2d_t *local_point,
                        const mppa_async_point2d_t *remote_point,
                        mppa_async_event_t *event)
{
  int status = 0;
  mppa_async_event_t local_event;
  mppa_async_event_t *_event = (event==NULL)? &local_event: event;
```

```
  off64_t remote_offset = offset +
                          (remote_point->xpos +
                          (remote_point->ypos*remote_point->xdim))*size;
  const char *local_address = (const char *)local +
                             (local_point->xpos +
                             (local_point->ypos*local_point->xdim))*size;
  // if extent is bigger than local, in any of dimensions, then return -1
  if ((width  > (local_point->xdim - local_point->xpos)) ||
      (height > (local_point->ydim - local_point->ypos))) {
    status = -1;
  } else {
    status |= mppa_async_sput_spaced(local_address, segment, remote_offset,
      width*size,                 /* size */
      height,                     /* count */
      local_point->xdim*size,     /* skew */
      remote_point->xdim*size,    /* space */
      _event);
  }
  if (event==NULL) {
    status = mppa_async_event_wait(_event);
  }
  return status;
}
```

7.3 Block 3D Asynchronous Transfers

7.3.1 mppa_async_point3d_t

```
typedef struct {
  int xpos;
  int ypos;
  int zpos;
  int xdim;
  int ydim;
  int zdim;
} mppa_async_point3d_t;
```

7.3.2 mppa_async_sget_block3d

```
/**
 * Start a 3D asynchronous transfer from the remote submatrix to the local submatrix.
 * local Local memory pointer where the data will be copied to.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the data will be copied from.
 * size Size in bytes of one data element.
 * width Width of the submatrix (x coordinate).
 * height Height of the submatrix (y coordinate).
 * local_point Point in the local submatrix.
 * remote_point Point in the remote submatrix.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_sget_block3d(void *local, const mppa_async_segment_t *segment,
                        off64_t offset, size_t size, int width, int height, int depth,
                        const mppa_async_point3d_t *local_point,
                        const mppa_async_point3d_t *remote_point,
                        mppa_async_event_t *event);
```

This function is a higher performance implementation of the following code:

```
int
```

```
mppa_async_sget_block3d(void *local, const mppa_async_segment_t *segment,
                        off64_t offset, size_t size, int width, int height, int depth,
                        const mppa_async_point3d_t *local_point,
                        const mppa_async_point3d_t *remote_point,
                        mppa_async_event_t *event)
{
  int status = 0;
  mppa_async_event_t local_event;
  mppa_async_event_t *_event = (event==NULL)? &local_event: event;
  off64_t remote_offset = offset +
                            (remote_point->xpos +
                            (remote_point->ypos*remote_point->xdim) +
                            (remote_point->zpos*remote_point->xdim *remote_point->ydim)
                                )*size;
  char *local_address = (char *)local +
                        (local_point->xpos +
                        (local_point->ypos*local_point->xdim) +
                        (local_point->zpos*local_point->xdim *local_point->ydim))*size;
  // if extent is bigger than local, in any of dimensions, then return -1
  if ((width  > (local_point->xdim - local_point->xpos)) ||
      (height > (local_point->ydim - local_point->ypos)) ||
      (depth  > (local_point->zdim - local_point->zpos))) {
    status = -1;
  } else {
    for (int i = 0; i < depth; i++){
      status |= mppa_async_sget_spaced(local_address, segment, remote_offset,
        width*size,                 /* size */
        height,                     /* count */
        local_point->xdim*size,     /* skew */
        remote_point->xdim*size,    /* space */
        _event);
      // sweep in Z direction
      remote_offset += remote_point->xdim*remote_point->ydim*size;
      local_address  += local_point->xdim*local_point->ydim*size;
    }
  }
  if (event==NULL) {
    status = mppa_async_event_wait(_event);
  }
  return status;
}
```

7.3.3 mppa_async_sput_block3d

```
/**
 * Start a 3D asynchronous transfer from the local submatrix to the remote submatrix.
 * local Local memory pointer where the data will be copied from.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the data will be copied to.
 * size Size in bytes of one data element.
 * width Width of the submatrix (x coordinate).
 * height Height of the submatrix (y coordinate).
 * local_point Point in the local submatrix.
 * remote_point Point in the remote submatrix.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_sput_block3d(const void *local, const mppa_async_segment_t *segment,
                        off64_t offset, size_t size, int width, int height, int depth,
                        const mppa_async_point3d_t *local_point,
                        const mppa_async_point3d_t *remote_point,
```

```
                          mppa_async_event_t *event);
```

This function is a higher performance implementation of the following code:

```
int
mppa_async_sput_block3d(const void *local, const mppa_async_segment_t *segment,
                          off64_t offset, size_t size, int width, int height, int depth,
                          const mppa_async_point3d_t *local_point,
                          const mppa_async_point3d_t *remote_point,
                          mppa_async_event_t *event)
{
  int status = 0;
  mppa_async_event_t local_event;
  mppa_async_event_t *_event = (event==NULL)? &local_event: event;
  off64_t remote_offset = offset +
                             (remote_point->xpos +
                             (remote_point->ypos*remote_point->xdim) +
                             (remote_point->zpos*remote_point->xdim*remote_point->ydim))*
                                 size;
  const char *local_address = (const char *)local +
                                (local_point->xpos +
                                (local_point->ypos*local_point->xdim) +
                                (local_point->zpos*local_point->xdim*local_point->ydim))*
                                    size;
  // if extent is bigger than local, in any of dimensions, then return -1
  if ((width  > (local_point->xdim - local_point->xpos)) ||
      (height > (local_point->ydim - local_point->ypos)) ||
      (depth  > (local_point->zdim - local_point->zpos))) {
    status = -1;
  } else {
    for (int i = 0; i < depth; i++){
      status |= mppa_async_sput_spaced(local_address, segment, remote_offset,
        width*size,                  /* size */
        height,                      /* count */
        local_point->xdim*size,      /* skew */
        remote_point->xdim*size,     /* space */
        _event);
      // sweep in Z direction
      remote_offset += remote_point->xdim*remote_point->ydim*size;
      local_address  += local_point->xdim*local_point->ydim*size;
    }
  }
  if (event==NULL) {
    status = mppa_async_event_wait(_event);
  }
  return status;
}
```

8 Global Synchronization

8.1 Remote Memory Operations

8.1.1 mppa_async_fence

```
/**
 * Remote memory fence (non-blocking) on a segment.
 * segment Asynchronous memory segment to fence.
 * event Event to wait for global completion, or NULL for blocking call.
 * return 0 on success.
 */
int
mppa_async_fence(const mppa_async_segment_t *segment, mppa_async_event_t *event);
```

8.1.2 mppa_async_peek

```
/**
 * Remote memory peek long long datum.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the datum will be operated.
 * result Pointer to store the result from the remote datum.
 * event Event to wait for global completion, or NULL for blocking call.
 * return 0 on success.
 */
int
mppa_async_peek(const mppa_async_segment_t *segment,
                off64_t offset, long long *result,
                mppa_async_event_t *event);
```

8.2 Remote Post Operations

8.2.1 mppa_async_poke

```
/**
 * Post a poke remote long long datum.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the datum will be operated.
 * value Value to be stored into the remote datum.
 * return 0 on success.
 */
int
mppa_async_poke(const mppa_async_segment_t *segment,
                off64_t offset, long long value);
```

8.2.2 mppa_async_postadd

```
/**
 * Post an atomic add to remote long long datum.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the datum will be operated.
 * addend Value to be atomically added to the remote datum.
 * return 0 on success.
 */
int
mppa_async_postadd(const mppa_async_segment_t *segment,
                   off64_t offset, int addend);
```

8.3 Remote Atomic Operations

8.3.1 mppa_async_fetchclear

```
/**
 * Start an atomic fetch-and-clear to a remote long long integer datum.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the datum will be operated.
 * result Pointer to store the atomic operation result.
 * event Event to wait for global completion, or NULL for blocking call.
 * return 0 on success.
 */
int
mppa_async_fetchclear(const mppa_async_segment_t *segment,
                      off64_t offset, long long *result,
                      mppa_async_event_t *event);
```

8.3.2 mppa_async_fetchadd

```
/**
 * Start an atomic fetch-and-add to a remote long long datum.
 * segment Identifies the remote memory segment.
 * offset Offset in segment where the datum will be operated.
 * addend Value to be atomically added to the remote datum.
 * result Pointer to store the atomic operation result.
 * event Event to wait for global completion, or NULL for blocking call.
 * return 0 on success.
 */
int
mppa_async_fetchadd(const mppa_async_segment_t *segment,
                    off64_t offset, int addend, long long *result,
                    mppa_async_event_t *event);
```

8.4 Local Condition Evaluation

8.4.1 mppa_async_cond_t

```
/**
 * Enumeration of asynchronous condition types.
 */
typedef enum {
  MPPA_ASYNC_COND_NE,
  MPPA_ASYNC_COND_EQ,
  MPPA_ASYNC_COND_LT,
  MPPA_ASYNC_COND_GE,
  MPPA_ASYNC_COND_LE,
  MPPA_ASYNC_COND_GT,
  MPPA_ASYNC_COND_LTU,
  MPPA_ASYNC_COND_GEU,
  MPPA_ASYNC_COND_LEU,
  MPPA_ASYNC_COND_GTU,
  MPPA_ASYNC_COND_ALL,
  MPPA_ASYNC_COND_NALL,
  MPPA_ASYNC_COND_ANY,
  MPPA_ASYNC_COND_NONE,
  MPPA_ASYNC_COND__
} mppa_async_cond_t;
```

8.4.2 mppa_async_evalcond

```
/**
 * Evaluate a condition between a local long long datum and a value.
```

```
 * local Remote memory pointer where the datum will be operated.
 * value Threshold value to be compared to the local datum.
 * event If not NULL, initialize for the deferred evaluation.
 * return 0 on success.
 */
int
mppa_async_evalcond(long long *local, long long value,
                    mppa_async_cond_t cond, mppa_async_event_t *event);
```

```
/**
 * Macros to specialize mppa_async_evalcond depending on cond.
 */
#define mppa_async_evalcond_ne(local, value, event) \
  mppa_async_evalcond(local, value, MPPA_ASYNC_COND_NE, event)
#define mppa_async_evalcond_eq(local, value, event) \
  mppa_async_evalcond(local, value, MPPA_ASYNC_COND_EQ, event)
#define mppa_async_evalcond_lt(local, value, event) \
  mppa_async_evalcond(local, value, MPPA_ASYNC_COND_LT, event)
#define mppa_async_evalcond_ge(local, value, event) \
  mppa_async_evalcond(local, value, MPPA_ASYNC_COND_GE, event)
#define mppa_async_evalcond_le(local, value, event) \
  mppa_async_evalcond(local, value, MPPA_ASYNC_COND_LE, event)
#define mppa_async_evalcond_gt(local, value, event) \
  mppa_async_evalcond(local, value, MPPA_ASYNC_COND_GT, event)
#define mppa_async_evalcond_ltu(local, value, event) \
  mppa_async_evalcond(local, value, MPPA_ASYNC_COND_LTU, event)
#define mppa_async_evalcond_geu(local, value, event) \
  mppa_async_evalcond(local, value, MPPA_ASYNC_COND_GEU, event)
#define mppa_async_evalcond_leu(local, value, event) \
  mppa_async_evalcond(local, value, MPPA_ASYNC_COND_LEU, event)
#define mppa_async_evalcond_gtu(local, value, event) \
  mppa_async_evalcond(local, value, MPPA_ASYNC_COND_GTU, event)
#define mppa_async_evalcond_all(local, value, event) \
  mppa_async_evalcond(local, value, MPPA_ASYNC_COND_ALL, event)
#define mppa_async_evalcond_nall(local, value, event) \
  mppa_async_evalcond(local, value, MPPA_ASYNC_COND_NALL, event)
#define mppa_async_evalcond_any(local, value, event) \
  mppa_async_evalcond(local, value, MPPA_ASYNC_COND_ANY, event)
#define mppa_async_evalcond_none(local, value, event) \
  mppa_async_evalcond(local, value, MPPA_ASYNC_COND_NONE, event)
```

8.5 Remote Queue Operations

8.5.1 mppa_async_enqueue

```
/**
 * Start asynchronous data enqueue on a remote queue.
 * segment Identifies the remote memory segment operated as a queue.
 * local Local base address of the data to enqueue.
 * size Size in bytes of the data to enqueue.
 * atomic If true, request for atomic queue (only works for small size).
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_enqueue(const mppa_async_segment_t *segment,
                   const void *local, size_t size, int atomic,
                   mppa_async_event_t *event);
```

8.5.2 mppa_async_dequeue

```
/**
 * Start asynchronous data dequeue from a remote queue.
 * Fails if the segment was not created locally as a queue.
 * segment Identifies the memory segment operated as a queue.
 * size Number of bytes to discard from the queue.
 * where Pointer to a pointer set to the address of the dequeued data.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_dequeue(mppa_async_segment_t *segment,
                   size_t size, void **where,
                   mppa_async_event_t *event);
```

8.5.3 mppa_async_discard

```
/**
 * Discard data from a remote queue.
 * Fails if the segment was not created locally as a queue.
 * segment Identifies the memory segment operated as a queue.
 * size Number of bytes to discard from the queue.
 * return 0 on success.
 */
int
mppa_async_discard(mppa_async_segment_t *segment,
                   size_t size);
```

9 Remote Memory Operations

9.1 Remote Memory Allocation

9.1.1 mppa_async_malloc

```
/**
 * Remote procedure call of the libc malloc function on the target segment.
 * segment Identifies the memory segment.
 * size Size in bytes to allocate.
 * result Pointer to the result of memory allocation.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_malloc(const mppa_async_segment_t *segment,
                  size_t size, off64_t *result,
                  mppa_async_event_t *event);
```

9.1.2 mppa_async_calloc

```
/**
 * Remote procedure call of the libc calloc function on the target segment.
 * segment Identifies the memory segment.
 * count Count of objects to allocate.
 * size Size in bytes per object to allocate.
 * result Pointer to the result of memory allocation.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_calloc(const mppa_async_segment_t *segment,
                  size_t count, size_t size, off64_t *result,
                  mppa_async_event_t *event);
```

9.1.3 mppa_async_realloc

```
/**
 * Remote procedure call of the libc realloc function on the target segment.
 * segment Identifies the memory segment.
 * offset The offset of memory to realloc.
 * size Size in bytes to allocate.
 * result Pointer to the result of memory allocation.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_realloc(const mppa_async_segment_t *segment,
                   off64_t offset, size_t size, off64_t *result,
                   mppa_async_event_t *event);
```

9.1.4 mppa_async_free

```
/**
 * Remote procedure call of the libc free function on the target segment.
 * segment Identifies the memory segment.
 * offset The offset of memory to free.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
```

```
int
mppa_async_free(const mppa_async_segment_t *segment,
                off64_t offset,
                mppa_async_event_t *event);
```

9.1.5 mppa_async_memalign

```
/**
 * Remote procedure call of the libc posix_memalign function on the target segment.
 * segment Identifies the memory segment.
 * alignment Buffer alignment.
 * size Number of bytes to allocate.
 * result Pointer to the result of memory alignment.
 * event If not NULL, initialize for the local completion, else blocking call.
 * return 0 on success.
 */
int
mppa_async_memalign(const mppa_async_segment_t *segment,
                    size_t alignment, size_t size, off64_t *result,
                    mppa_async_event_t *event);
```

9.1.6 mppa_async_address

```
/**
 * Convert a segment offset into an address.
 * segment Pointer to the segment object in local memory.
 * offset Offset in segment to convert to address.
 * result Pointer to store the address corresponding to offset.
 * return 0 on success.
 */
int
mppa_async_address(const mppa_async_segment_t *segment,
                   off64_t offset, void **result);
```

Bibliography

[1] Karl Feind. Shared Memory Access (SHMEM) Routines. Cray User Group Proc., 1995.

[2] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI (2nd ed.): portable parallel programming with the message-passing interface*. MIT Press, 1999.

[3] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the 1994 Winter USENIX Conference*, pages 115–131, 1994.

[4] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High performance remote memory access communication: The armci approach. *Int. J. High Perform. Comput. Appl.*, 20(2):233–253, May 2006.

A Asynchronous Operations Library Examples

B Asynchronous Operations Library Usage

B.1 IO Cluster Side

The IO cluster should not be used in this configuration as compute clusters are master of the application in the remote memory. The IO executable needs to be linked with libmppa_async.a. In Kalray Makefiles at link time on the IO cluster:

```
your_io_executable_name-lflags += -lmppa_async
```

B.2 Compute Cluster Side

The compute cluster executable needs to be linked with libmppa_async.a. In Kalray Makefiles at link time on the compute cluster executable:

```
your_compute_cluster_executable_name-lflags += -lmppa_async
```

B.3 Internal Dependencies

This library has internal dependencies which are `libmppanoc.a`, `libmpparouting.a` and `libmppapower.a`. In Kalray Makefiles at link time on both IO cluster and compute clusters:

```
your_io_executable_name-lflags += -lmppapower -lmppanoc -lmpparouting
your_compute_cluster_executable_name-lflags += -lmppapower -lmppanoc -lmpparouting
```