# A Dataflow Communications Library for Adapteva's Epiphany

Sebastian Raase,
Halmstad University, Sweden,
`sebastian.raase@hh.se`

December 2015

Adapteva's Epiphany platform is a scalable low-power manycore architecture. Even though Adapteva provides an ANSI C compatible compiler and runtime as well as a Software Development Kit (eSDK), developing for this platform is not particularly easy.

At Halmstad University, we are interested in dataflow applications and have developed a suitable communications library (*e-commlib*) for the Epiphany, which we would like to release under a permissive 2-clause BSD license.

Given sufficiently aware compute kernels, e-commlib projects can also be compiled and run in a Linux-pthreads environment, which simplifies both development and (functional) debugging.

This Technical Report shall document both e-commlib (version 3) and our surrounding infrastructure.

# Contents

# 1 Introduction

This report describes *e-commlib v3*, a channel-based communications library for the Adapteva Epiphany platform developed at Halmstad University. It is primarily intended as documentation of the library as well as the environment it is designed to operate in.

## 1.1 Background

Adapteva's Epiphany platform has become available to a wide audience due to the Parallella development board and the related 2012 Kickstarter project. The Parallella design is considered the reference design for the Epiphany architecture and has been used when developing *e-commlib*. Although Adapteva provides a fully open source Software Development Kit (eSDK) based on the GNU toolchain, programming the Epiphany system is not particularly easy.

As part of our ongoing work with dataflow applications in the ESCHER [1] and HiPEC [2] projects, we have developed *e-commlib* as a basic communication library suitable for our needs. Since our solutions turns out to be rather generic, we have decided to publish it under a permissive 2-clause BSD license in order to provide a basic infrastructure for others working with the Epiphany platform.

## 1.2 Adapteva Epiphany and Parallella

Current Epiphany platforms are heterogeneous systems. In addition to the Epiphany chip, they provide a host processor running Linux and an FPGA implementing the eLink communication protocol. On the common Parallela boards, a Xilinx Zynq 7010/7020 is paired with an Epiphany-III E16G3, as shown in Figure 1.1. Although the Epiphany contains four eLink interfaces, only one of them is connected to the Zynq. On the Parallella, a 32 MB area of the 1 GB memory is mapped to be accessible by both the host system and the Epiphany. This memory is generally called *shared memory*.

The Epiphany architecture itself is a low-power scalable, parallel computing fabric, consisting of a two-dimensional array of compute nodes connected by a mesh network-on-chip (NoC) operating in a shared address space. The upper 12 bits of an address specify row and column of any given node; by definition, the local node is always visible at row and column zero, as well. Each node contains a RISC CPU with 32 KB of high-speed SRAM and additional peripherals (DMA engines, event timers).

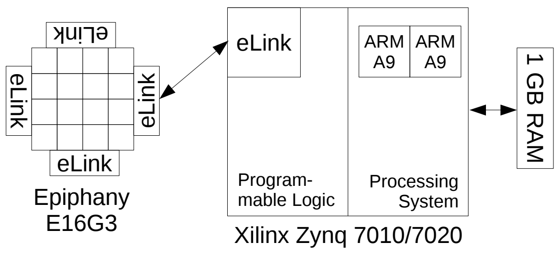

Figure 1.1: Parallella System Overview

The NoC is made out of three meshes with different purposes. Read requests travel through the *rMesh*, while *cMesh* and *xMesh* carry write transactions destined for on-chip and off-chip nodes, respectively. To the application, off-chip traffic is indistinguishable from on-chip traffic, apart from lower bandwidth and higher latency. Routing uses an XY algorithm, with round-robin arbitration at each intersection.

Transactions to foreign nodes follow a weakened memory-order model. Write transactions happen asynchronously, and this latency is exposed to the software (e.g. reading back a recently written value may return the old value). Software must be prepared to handle this correctly. Also, reading a foreign address is at least 16x slower than writing while also incurring high latency.

For more details about the Epiphany architecture, please refer to [4].

## 1.3 Pthreads

POSIX Threads (pthreads) are a standardized [5] way to implement multi-threading on POSIX-based operating environments, for example on Linux. While pthreads is a supported target for *e-commlib*, it is not to be considered production-grade. Instead, a reasonably similar, but by no means identical environment is provided to simplify development and (function) debugging of computing kernels for the Epiphany.

Modern computers tend to be much faster than the Parallella, reducing compilation and turnaround times; inside the pthreads environment, all host features (large memories, direct file and terminal I/O, debuggers, etc.) are available.

Kernels must be written to accomodate the environmental differences in order to work correctly in both environments. Compilation and linking for the Epiphany target is done per-kernel, but the pthreads target produces a single, unified binary. As an example, per-kernel variables must be global in the former, but marked thread-local (`__thread`) in the latter case to behave similarly. Other related issues (e.g. project-global vs. kernel-local namespaces, etc.) must be taken into account as well.

## 1.4 Motivation

Adapteva provides a software development kit for the Epiphany, called eSDK. Together with the toolchain, this kit also contains a few helper utilities, a programming API and some example. However, these interfaces operate on a quite low level and do not provide higher abstractions. Instead, they focus mostly on driving the hardware.

Since we work with dataflow problems and languages, we needed a communication framework for the Epiphany platform, which the eSDK does not provide.

## 1.5 Design Goals

The main design of *e-commlib* was driven by these goals:

- ease of use, both for hand-written and generated code
- usable as basis for our CAL (CAL Actor Language) compiler framework [3]
- low runtime overhead
- additional channel types easy to implement
- unidirectional, synchronous communication channels
- static, table-driven per-channel configuration

# 2 User Documentation

Adding *e-commlib* to a project is relatively straightforward. The implementation is purely done in C and split across only four files:

- `commlib_cfg.h`: configuration file
- `commlib.h`: header file
- `commlib.c`: implementation
- `commlib-host.c`: host-side channel support (optional)

Both header files are shared between the host and device (Epiphany) toolchains, and it must be ensured that the in-memory data layouts are compatible. Also, it is strictly assumed that both host and device use a 32-bit pointer type and are little-endian; on x86_64 hosts, a 32-bit multilib environment is required.

An example project suitable for both the Epiphany and pthreads environments is provided with this report.

Identifiers related to *e-commlib* are generally prefixed with either `comm_` or `COMM_` in order to avoid namespace pollution.

## 2.1 Building

To build *e-commlib*, you need to define either `COMM_EPIPHANY` or `COMM_PTHREADS`, depending on your chosen target. Please note that compute kernels must also be aware of the differences in order to run in both environments.

The host code needs to `#include "commlib.h"` to create the channel descriptor table. In order to use `HOST` channels, the file `commlib-host.c` must be included in the project as well. On the Epiphany target, each core needs to be linked against `commlib.o`.

Building the example project for pthreads has been tested on Linux (armel and x86_64 with 32-bit multilib). Building for Epiphany has been tested on a Parallella (using eSDK version 2015.1). For the example program, the target selection is done exclusively in the Makefile.

## 2.2 Compile-time configuration

The file `commlib_cfg.h` contains the compile-time configuration for *e-commlib*. Here, different features can be enabled or disabled. Most configuration options are boolean flags, which are `#define`'d to enable or `#undef`'d to disable.

Currently, the following configuration options exist:

- `#define COMM_NUM_CHANNELS`
  This *integer* constant defines the maximum number of channels supported in the system. Declaring more channels than needed just wastes a bit of memory in the descriptor table. **Must be provided.**
- `#define COMM_CFG_CTYPE_DEFAULT`
  Enables support for `DEFAULT` channels, which are the basic channel type to connect cores with each other. Increases code size by about 1 KB.
- `#define COMM_CFG_CTYPE_HOST`
  Enables support for `HOST_INPUT` and `HOST_OUTPUT` channels, which are used to associate communication channels with files on the host. Include `commlib-host.c` into your host build system to use. Increases code size by about 1 KB.
- `#define COMM_CFG_USE_IDLE`
  Enables the `IDLE` framework, which replaces busy waiting for blocking operations. On the Epiphany, blocking will be implemented by calls to IDLE and reads or writes will trigger a Software Interrupt (SWI) on the remote core to wake it up. On pthreads, blocked threads will just call `sched_yield()`.
- `#define COMM_CFG_USE_MALLOC`
  By default, *e-commlib* provides its own memory allocator, which uses a user-provided heap. Enabling this option makes it use the system implementation of `malloc()` instead. **Does not work on Epiphany.**

## 2.3 Channel declarations

All channels in the system are declared using channel descriptors and channels only operate on fixed-size tokens. Each descriptor describes a single channel and contains information about its type, its source and destination, the size of each token and the number of tokens this channel can buffer.

To get started, create an array of type `comm_channel_t[COMM_NUM_CHANNELS]`, which needs to be made available to all cores. This descriptor table is only accessed at initialization time and when requesting access handles, so the runtime impact of putting it in shared memory (recommended) is small. Please note that this table is written to at initialization time and that writes done by one core need to be visible to the other cores. For each channel type, initializer macros are provided to conveniently fill in the table.

### 2.3.1 `DEFAULT` channels

For `DEFAULT` channels, only four parameters are needed: two core numbers for source and destination, token size in bytes and buffer size in tokens.

While it is possible to read or write multiple tokens in a single functions call, such requests will be broken down to single-token accesses to avoid deadlocks. The time needed to pass small tokens around is small compared to the overhead introduced by the indirect function calls, so consider using large tokens instead of many small ones.

### 2.3.2 `HOST_INPUT` and `HOST_OUTPUT` channels

`HOST_INPUT` and `HOST_OUTPUT` initializer macros are implicitly tied to the host program implementation. They assume a fixed-size `typedef struct {...} shm_t;` located at the beginning of shared memory, which contains a sufficiently-sized field to be used for the channel data structure and token buffer.

The first three parameters are source and destination (file name or core number, depending on the channels direction) and the field name inside `shm_t` to use as the buffer. The table itself only stores the buffer offset from the beginning of shared memory.

The file `commlib-host.c` implements the host application part of host channels.

Please note the following:

- Specifying a `HOST_OUTPUT` channel with the magic file name *stdout* will connect it to the file descriptor `1` (stdout) instead of creating a file.
- Input files must exist. Output files will be automatically created. Existing output files will be truncated before use. Special files (e.g. named pipes) stay special.
- Using the same file in both an output and input channel works. Define the output channel first to avoid errors in case of non-existing files.
- Using the same file in multiple input channels works; channels read the same data. Using the same file in multiple output channels (including *stdout*) results in undefined behaviour.
- To avoid wearing out the SD card on the Epiphany, consider using network or tmpfs mounts (e.g. `/dev/shm/`) for intermediate files.

### 2.3.3 Example

The following example describes the dataflow structure shown in Fig. 2.1. It connects two files to a one-dimensional processing chain, avoiding cross-traffic. On-chip channels can store up to four 36-byte tokens in their buffers; off-chip channels store up to 128 tokens. The channel indices, which are needed to acquire communication handles are given as comments in the table below.

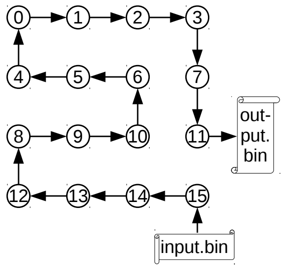

Figure 2.1: Example Dataflow Structure

```
/* in commlib_cfg.h */
#define COMM_NUM_CHANNELS 17

/* in host code */
#define TOKEN_NUM  4
#define TOKEN_SIZE (9 * sizeof(float))
comm_channel_t channels[COMM_NUM_CHANNELS] = {
    HOST_INPUT("input.bin", 15, input_buf, 128, TOKEN_SIZE),    /*  0 */
    DEFAULT(15, 14, TOKEN_NUM, TOKEN_SIZE),                     /*  1 */
    DEFAULT(14, 13, TOKEN_NUM, TOKEN_SIZE),                     /*  2 */
    DEFAULT(13, 12, TOKEN_NUM, TOKEN_SIZE),                     /*  3 */
    DEFAULT(12,  8, TOKEN_NUM, TOKEN_SIZE),                     /*  4 */
    DEFAULT( 8,  9, TOKEN_NUM, TOKEN_SIZE),                     /*  5 */
    DEFAULT( 9, 10, TOKEN_NUM, TOKEN_SIZE),                     /*  6 */
    DEFAULT(10,  6, TOKEN_NUM, TOKEN_SIZE),                     /*  7 */
    DEFAULT( 6,  5, TOKEN_NUM, TOKEN_SIZE),                     /*  8 */
    DEFAULT( 5,  4, TOKEN_NUM, TOKEN_SIZE),                     /*  9 */
    DEFAULT( 4,  0, TOKEN_NUM, TOKEN_SIZE),                     /* 10 */
    DEFAULT( 0,  1, TOKEN_NUM, TOKEN_SIZE),                     /* 11 */
    DEFAULT( 1,  2, TOKEN_NUM, TOKEN_SIZE),                     /* 12 */
    DEFAULT( 2,  3, TOKEN_NUM, TOKEN_SIZE),                     /* 13 */
    DEFAULT( 3,  7, TOKEN_NUM, TOKEN_SIZE),                     /* 14 */
    DEFAULT( 7, 11, TOKEN_NUM, TOKEN_SIZE),                     /* 15 */
    HOST_OUTPUT(11, "output.bin", output_buf, 128, TOKEN_SIZE), /* 16 */
}
```

## 2.4 Usage

### 2.4.1 Initialization

Before *e-commlib* can be used, the `comm_init()` function must be called. Each kernel must call this function and provide both a pointer to the descriptor table as well as its identifier (core number, as in the descriptor table). Since channels are instanciated at runtime according to the descriptor table, a heap is required. It is recommended to statically allocate a block of memory and provide both address and size of it when calling the function; if configured, the system heap can be used instead. Initialization may block until all cores have initialized successfully.

Channels are always accessed through handles of the opaque type `comm_handle_t`. These are obtained by calling `comm_get_rhandle()` (for reading) or `comm_get_whandle()` (for writing), depending on the channel's direction. These functions take the descriptor table *index* of the specified channel. Note that read and writes handles generally differ.

#### Examples

Using the descriptor table shown above, core 14 is configured to use the channels 1 (for reading) and 2 (for writing). The initialization could look like this:

```
comm_init(shm.table, 14, &heap14[0], sizeof(heap14));
comm_handle_t input  = comm_get_rhandle(1);
comm_handle_t output = comm_get_whandle(2);
```

Core 15 would use the channels 0 (for reading) and 1 (for writing) instead. Please note that the kernel code is not aware of the different channel types used in the table:

```
comm_init(shm.table, 15, &heap15[0], sizeof(heap15));
comm_handle_t input  = comm_get_rhandle(0);
comm_handle_t output = comm_get_whandle(1);
```

### 2.4.2 Reading

Reading handles support three access methods:

- `int comm_read(comm_handle_t handle, void *buf, size_t n)`
  Read `n` tokens from the channel `handle` into the buffer pointed to by `buf` and consume them. This function blocks until all `n` tokens have been read and returns the number of tokens read.
- `int comm_peek(comm_handle_t handle, void *buf, size_t n)`
  Read up to `n` tokens from the channel `handle` into the buffer pointed to by `buf`, but do not consume them. This function does not block and returns the number of

tokens copied into the buffer. A subsequent call to `comm_read()` or `comm_peek()` will return the same tokens again.

- `int comm_level(comm_handle_t handle)`
  Returns the number of available tokens for channel `handle`. Calls to `comm_read()` asking for this number of tokens (or less) are guaranteed to not block.

### 2.4.3 Writing

Write handles support only two access methods:

- `int comm_write(comm_handle_t handle, void *buf, size_t n)`
  Writes `n` tokens from the buffers pointed to by `buf` into the channel `handle`. The function blocks until all `n` tokens have been written and returns the number of tokens written.
- `int comm_space(comm_handle_t handle)` Returns the amount of free space in the channel `handle`. Calls to `comm_write()` trying to write this number of tokens (or less) are guaranteed not to block.

### 2.4.4 Error Handling

Calling any *e-commlib* function may result in an error, which is always fatal.

Three numeric error codes are used:

- `TRAP_OOM` (value 50): Out of memory.
  The configuration requires more dynamic memory than what is available. Try to increase the heap size for the specific core or change the channel configuration.
- `TRAP_TABLE` (value 51): Invalid table entry or index.
  Either the descriptor table contains invalid data (usually caused by stale object files after changing the configuration), or the core tried to request an invalid handle.
- `TRAP_INVALID` (value 52): Invalid access function.
  An unsupported access function was called (e.g. reading through a write-handle).

If an error occurs, *e-commlib* will execute the corresponding `TRAP` instruction on the Epiphany target, halting the affected core. For the pthreads target, the application will exit with an error message immediately.

# 3 Technical Documentation

Although *e-commlib* is written in C, it follows some object orientation principles. In order to provide extensibility, a generic channel base class is used, which is extended by each channel implementation. The runtime overhead of using this system consists of an additional indirect call per request, and is considered negligible.

However, doing most of the initialization at runtime results in a quite large, static footprint. The communication library needs about 1 KiB of code, and each enabled channel type adds another 1 KiB. The data size depends almost exclusively on the channel configuration for each core.

## 3.1 Memory Management

Generally, dynamic memory is implemented in the C library. On the Epiphany, the heap is located in shared memory by default, which is unfortunate when implementing an inter-core communication facility.

In *e-commlib*, dynamic memory is required only for initialization, and a small memory allocator suitable for these purposes is provided. This allocator, called `comm_malloc()` can use a statically allocated block of memory to implement a heap and is initialized at the time `comm_init()` is called.

All allocations handled by this allocator are aligned to an eight-byte boundary and can not be freed. No additional padding is done between allocations.

When `COMM_CFG_USE_MALLOC` is defined at compile time, this allocator is replaced by calls to the C library memory allocator instead; in this case, the heap specification given at initialization time is ignored. However, this is not recommended.

## 3.2 Descriptor Table

The data structure used for a descriptor table entry is shown in Tab. 3.1. The type field contains a number denoting the channel type, which depends on the configuration; the value zero is reserved for invalid table entries.

The source and destination fields are tuples consisting of a core number (which needs to match the corresponding number given to `comm_init()`), a device pointer (`dptr`) and a host pointer (`hptr`).

| Name | Description |
|---|---|
| `type` | Channel Type |
| `src` | Source Core |
| `dst` | Destination Core |
| `tsize` | Token Size (Bytes) |
| `tnum` | Channel Size (Tokens) |

Table 3.1: Channel Descriptors

At initialization time, the device pointers are set to NULL, and are set by the `comm_init()` function as part of the initialization sequence. They point to a channel-specific data structure. The host pointers are only used by host channels and allow the host code to store additional per-channel data.

## 3.3 Channels

All channel implementations inherit from a channel base class called `comm_data_t` (shown in Tab. 3.2). It contains the type of channel, the token and buffer sizes for this channel and function pointers to each access function. Some access functions may be unavailable; trying to call them raises a `TRAP_INVALID` error.

| Name | Description |
|---|---|
| `type` | Channel Type |
| `tsize` | Token Size (Bytes) |
| `tnum` | Channel Size (Tokens) |
| `*readfn` | Read Function |
| `*peekfn` | Peek Function |
| `*writefn` | Write Function |
| `*levelfn` | Level Function |
| `*spacefn` | Space Function |

Table 3.2: Channel Base Class

By having access function calls being indirect, channel differences are abstracted away at a small runtime cost. However, requiring a different set of access functions for each channel type increases the code size footprint.

### 3.3.1 `DEFAULT` channels

`DEFAULT` channels are FIFOs, implemented as ring buffers. On the Epiphany, non-local writes are much faster than reads. Also, reads and writes to remote addresses are not strongly ordered. Writes to the same destination are strongly ordered, however.

Each end of the communication channel consists of a data structure, which extends the base class and contains a pointer to the other end of the channel. Both ends also contain a pointer to the buffer, which is always located on the destination core.

| **Source Core** | **Destination Core** | **Description** |
|---|---|---|
| `data` | `data` | Channel Base Class (see Table 3.2) |
| `*dst` | `*src` | Remote End |
| `rp` | `rp` | Read Index |
| `-` | `pp` | Peek Index |
| `wp` | `wp` | Write Index |
| `*buf` | `*buf` | Pointer to buffer |

Table 3.3: `DEFAULT` channel data structures

The ring buffers are implemented by using read and a write indices, which are used to point into the buffer. The buffer itself always keeps one additional unused element to avoid ambiguities (if `rp == wp`). While the read index is located on the destination core, the write index is located on the source core; the respective other core contains a shadow copy. This way, cores never read from remote memory.

The peek index is only used on the destination core to keep track of the number of tokens peeked. Since a call to `comm_peek()` does not influence the state of a channel, this index is not shadowed on the source core.

On `DEFAULT` channels, the buffer is always accessed through the `memcpy()` library function. After the buffer has been accessed, first the related local data structures are updated, then the remote data structures are written. This order avoids synchronization issues between the two cores sharing the buffer.

### 3.3.2 `HOST` channels

`HOST` channels are implemented twice, once in the host program (`commlib-host.c`) and once in the main library (`commlib.c`). These channels always keep the buffer in shared memory and use the `memcpy()` library function access it. While these channels have been introduced as `HOST_INPUT` or `HOST_OUTPUT` channels earlier, they only differ in direction and are implemented as a single `HOST` channel type.

The implementation of `HOST` channels is very similar to the implementation of `DEFAULT` channels on the device side. However, instead of storing the shadow indices on the remote end, they are stored inside local memory as well (see Tab. 3.4).

In shared memory, only a minimal data structure (shown in Tab. 3.5) is stored. The buffer should be guaranteed to be eight-byte aligned.

Depending on the channel direction, the host is either authoritative for the read index (`HOST_OUTPUT` channels) or the write index (`HOST_INPUT` channels).

| Name | Description |
|---|---|
| `data` | Channel Base Class (see Table 3.2) |
| `rp` | Read Index |
| `pp` | Peek Index |
| `wp` | Write Index |
| `*rpp` | Read Index Shadow |
| `*wpp` | Write Index Shadow |
| `*buf` | Pointer to buffer |

Table 3.4: `HOST` channel data structures - local memory

| Name | Description |
|---|---|
| `rp` | Read Index |
| `wp` | Write Index |
| `buf[]` | Buffer |

Table 3.5: `HOST` channel data structures - shared memory

Since these channels are supposed to be connected to files on the host file system, the host application library stores an additional, host-only data structure containing the file name, the file descriptor and a token counter. The latter counts tokens read from or written to the attached file and is useful to provide status updates at runtime.

### 3.3.3 `IDLE` support

Trying to read from an empty buffer or trying to write to a full buffer will block. By default, blocking is implemented through busy waiting in order to achieve maximum performance. When `COMM_CFG_USE_IDLE` is defined at compile time, the `IDLE` support is enabled instead. While `HOST` channels are not affected by this change at all, the behaviour of `DEFAULT` channels will change.

On the Epiphany target, an empty interrupt handler for the Software Interrupt (`SWI`) is defined and the `SWI` is enabled. Now, instead of busy waiting, the corresponding Epiphany core will execute the `IDLE` instruction instead and conserve power. Also, whenever data is read from or written to a channel, a Software Interrupt is triggered on the remote core to (possibly) wake it up.

While these interrupts introduce additional runtime overhead, they may reduce power consumption. More importantly, since the Epiphany hardware event timers support counting of idle clock cycles, the `IDLE` support provides a good, low-overhead estimate for the amount of time cores spend waiting for each other.

On the Pthreads target, enabling the `IDLE` support will result in `sched_yield()` calls while a thread is blocked.

# 4 Conclusion

We have implemented a channel-driven dataflow communications library for Adapteva's Epiphany system. While designed for the Epiphany, this library can also be used in a Linux-pthreads environment, which can simplify code implementation and debugging compared to the environment provided by Adapteva's current eSDK.

This library tries to be easy to use and integrate into existing projects, while keeping the runtime overhead low. At the same time, the design allows easy implementation of additional channel types to experiment on and explore the Epiphany platform.

Due to its design, *e-commlib* is quite generic and should be relatively easy to port to other shared-memory architectures, even if they only provide weakened memory-ordering guarantees; the pthreads target reflects this possibility. However, the generic design has its price. With all features enabled, *e-commlib* adds slightly less than 3 KiB of code to each Epiphany core and the runtime overhead for small tokens should not be neglected.

Despite its shortcomings, we release this library under a permissive 2-clause BSD license in the hope that people will find it useful.

# Bibliography

[1] ESCHER project: Embedded Streaming Computations on Heterogeneous Energy-efficient aRchitectures. `http://ceres.hh.se/mediawiki/ESCHER`.

[2] HiPEC project: High Performance Embedded Computing. `http://hipec.cs.lth.se/`.

[3] Essayas Gebrewahid, Mingkun Yang, Gustav Cedersjo, Zain Ul Abdin, Veronica Gaspes, Jörn W Janneck, and Bertil Svensson. Realizing efficient execution of dataflow actors on manycores. In *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, pages 321–328. IEEE, 2014.

[4] Andreas Olofsson, Tomas Nordström, and Zain Ul-Abdin. Kickstarting High-performance Energy-efficient Manycore Architectures with Epiphany. 48th Asilomar Conference on Signals, Pacific Grove, CA, Nov. 2-5, 2014.

[5] The IEEE and The Open Group. Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition. `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html`.