Cell-MPI

Mastering the Cell Broadband Engine architecture through a Boost based parallel communication library

Sebastian Schaetz, Joel Falcou, Lionel Lacassagne

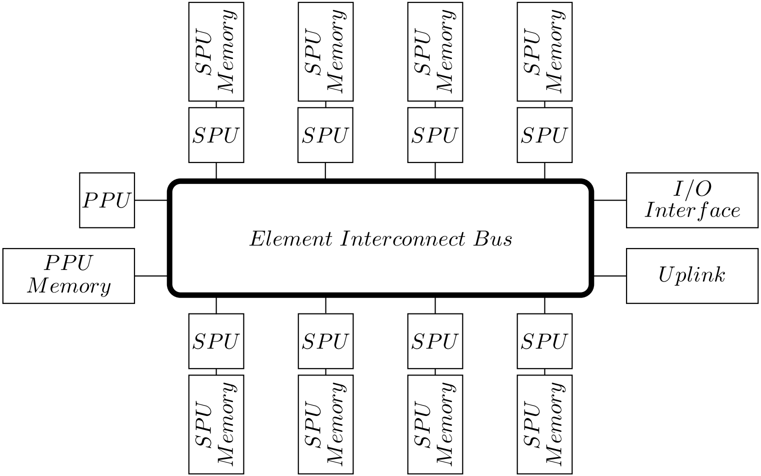
Digiteo Foundation, LRI - University Paris South XI, CEA LIST

May 17, 2011

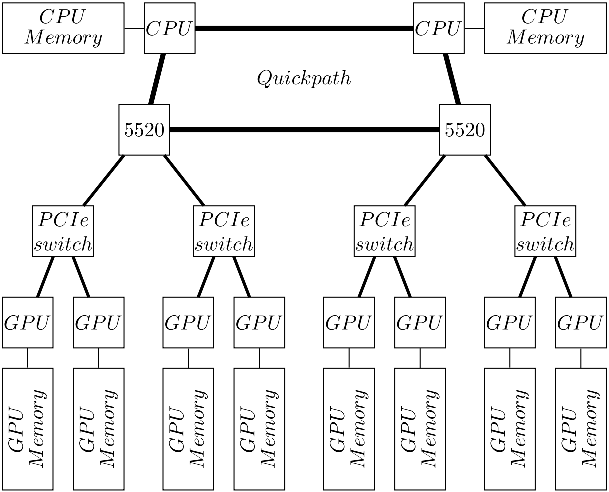
Why a talk about a library for the Cell in 2011

- Heterogeneous and composable architectures are not uncommon, powerful and worth studying.
- We present useful concepts that apply to all of them.
- We illustrate the lessons we learned as we used Boost libraries on a constricted platform and
- elaborate what choices we had to make and why we made them as we created a Boost-like library for this platform.

Cell Broadband Engine - Schematic



A similar architecture - Multi-GPU Schematic



Cell Broadband Engine - The good stuff

- Power architecture core paired with up to 8 streamlined vector co-processors: 204.8 GFlops/s (single) 102.4 GFlops/s (double)
- High data transfer bandwidth: theoretical 204.8 GB/s
- Good performance/watt (0.87 double precision GFlops/s per Watt for IBM BladeCenter QS22)

Due to these advantages, the CBE is a good fit for multimedia and vector processing applications as well as scientific computation.

Cell Broadband Engine - The bad stuff

- Distributed system on one chip, explicit communication necessary
- SPE Memory limitations
  - 256kB for code and data per SPE
  - no overflow detection
- Communication intricacies
  - packet size
  - address alignment
  - explicit DMA
- Optimization for speed
  - SIMD (assembler-like)
  - convoluted pipeline mechanism

Due to these restrictions, the complexity of programming the CBE is comparable to writing code for embedded systems.

Writing code for the CBE

- PPE and SPE entry points in separate main functions
- Compilers: ppu-gcc, spu-gcc
- SPE object file passed to ppu-embedspu to generate library

  exports symbol that is accessible from PPE code

- PPE creates thread for each SPE and loads the symbol
- Argument passed to thread is accessible in SPE through argument vector
- Usual approach: argument is pointer to structure in main memory; structure is loaded to SPE through explicit DMA call:

```
/* DMA control block information from system memory. */
mfc_get((void*)&parms, parm_ptr, (sizeof(parms)+15)&~0xF, tag, td, rd);
mfc_write_tag_mask(1<<tag);
mfc_read_tag_status_all(); /* Wait for DMA to complete */
```

Writing code for the CBE - continued

- "Getting started" can be tedious when developing for the CELL since compilation procedure and startup are not trivial

- CMake to the rescue:
  - Great tool to simplify basically any build-related steps
  - Find all required libraries and binaries on the system
  - Low-level macros: `ACTIVATE_PPE_COMPILER()`, `ACTIVATE_SPE_COMPILER()`
  - `ADD_SPE_MODULE(target symbol file0 file1 ... fileN)`

- C++ and Boost to the rescue:
  - Wrap recurring boilerplate code in clearly laid out functions and classes
  - A kernel function should be declared and behave like a free function

Cell-MPI Bootstrapping

- Launching a kernel function passing a data structure

  `struct mydatastruct { int x; int y; int z; };`

- Kernel is defined with:

```
BEGIN_CELL_KERNEL()
{
  mydatastruct * ptr;
  SPE_Custom(ptr);
  RETURN((ptr->x + ptr->y) * ptr->z);
}
END_CELL_KERNEL()
```

- In PPE code the kernel is registered with

  `PPE_REGISTER_KERNEL(kernel);`

- The runtime is initialized with `PPE_Init();`

Cell-MPI Bootstrapping - continued

- The kernel is then called asynchronously:

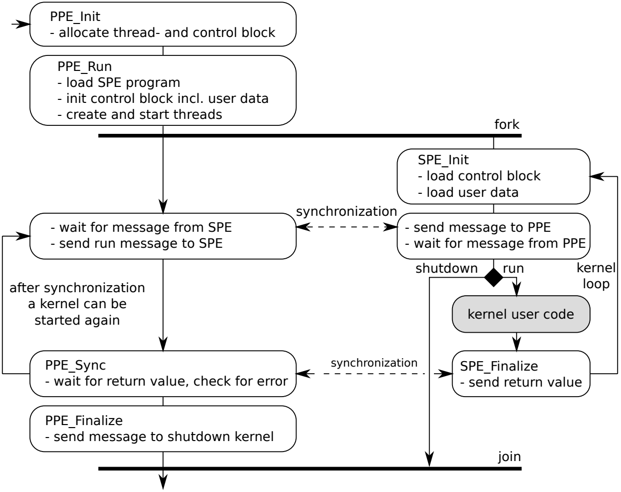
```
mydatastruct mydata(1, 5, 7);
PPE_Run(kernel, mydata);
```

- The PPE can wait for kernel completion: `PPE_Sync();`
- and access the kernels return value:

```
int returnvalues[CBE_MPI_NUM_SPE];
PPE_Return(&returnvalues[0]);
```

- The runtime is finalized with `PPE_Finalize();`

Cell-MPI Bootstrapping Mechanism



Cell-MPI Bootstrapping - Boostified

- A kernel can be declared in both PPE and SPE code with:

```
SPE_FUNCTION(kernel_, kernel, (int x) (int y) (int z) );
```

- and implemented as a free function in SPE code

```
int kernel(int x, int y, int z)
{
  return (x+y) * z;
}
```

- It can then be called as a free function from PPE code:

```
int * returnvalues = kernel(2, 5, 7);
```

- or asynchronously:

```
kernel_async(2, 5, 7);
PPE_Sync();
```

So we do C++ but...

The architecture forces some restrictions especially on the SPE part of the library:

- Compilation without run-time type information
- No dynamic memory allocation for predictable footprint
- Custom, lightweight STL compatible allocators
- Exception handling deactivated

Exception emulation

Due to architecture limitations we emulate exceptions:

- An exception stops the kernel and notifies the PPE
- Only an error code is "thrown":

```
#define THROW(errno) {spe_errno = errno ; SPE_Finalize(-1); exit(0);}
```

- The PPE translates the error code into real exceptions:

```
struct spe_error_bundle
{ std::vector<spe_error_data> exception_info; };
```

```
typedef boost::error_info<struct tag_spe_error_info_bundle,
   spe_error_bundle> spe_error_info_bundle;
```

```
struct spe_runtime_exception : virtual boost::exception {};
```

Exception emulation - continued

- If desired SPE exceptions can interrupt PPE execution
- To define errors we use the same trick as in boostified bootstrapping:

```
ERROR(MPI_TAG_MISMATCH, 7, "Send receive tag mismatch")
ERROR(BOOST_FUNCTION_BAD_CALL, 12, "Bad boost function call")
ERROR(BAD_ALLOC, 14, "bad alloc")
```

- Compiled with the SPE compiler (`#ifdef _SPE_`) generates:

```
enum { MPI_TAG_MISMATCH = 7, BOOST_FUNCTION_BAD_CALL = 12,
       BAD_ALLOC = 14 };
```

- And with the PPE compiler generates a vector of objects:

```
struct spe_error_struct
{ int id; const char * symbol; const char * message; };
```

Unit Testing

- Boost.Test is great but builds don't fit SPEs:
  `libboost_unit_test_framework.so.1.45.0: 998kB`
- First idea: `boost/detail/lightweight_test.hpp`
  misses a lot of the Boost.Test goodness

Enter SPE-Unit:

- Compromise between lightweight and feature-complete
- Designed after Boost.Test

Unit Testing - SPE Unit Features

- Only one test suite is available: `CBE_MPI_SPEUNIT_AUTO_TEST_SUITE();`
- Tests are started explicitly:

```
uint32_t result = CBE_MPI_SPEUNIT_RUN_TEST_SUITE();
SET_RETURN_VALUE(result);
```

- The powerfull `AUTO_TEST_CASE_TEMPLATE(testname, T, typelist)` and a normal template `TEST_CASE_TEMPLATE(testname)` are included
- Different test tool levels are supported: `WARN_*`, `CHECK_*`, `REQUIRE_*`
- Strings can be disabled to reduce overhead (silent mode)
- Emulated SPE exceptions can be validated with test tools like `CBE_MPI_REQUIRE_THROW`

Unit Testing - Example

```
typedef boost::mpl::vector_c<int,1,2,4,8,16> aligned_alloc_alignments;

CBE_MPI_SPEUNIT_AUTO_TEST_SUITE();

CBE_MPI_SPEUNIT_AUTO_TEST_CASE_TEMPLATE( aligned_malloc_free_test, T,
  aligned_alloc_alignments )
{
  aligned_ptr<void,T::value> ptr = aligned_malloc<T::value>(T::value);
  CBE_MPI_SPEUNIT_REQUIRE_EQUAL(is_aligned<T::value>(ptr.get()),true);
  cbe_mpi::aligned_free(ptr);
  CBE_MPI_SPEUNIT_REQUIRE_EQUAL(ptr.get(),((void*)(0)));
}

int kernel(void)
{
  uint32_t result = CBE_MPI_SPEUNIT_RUN_TEST_SUITE();
  SET_RETURN_VALUE(result);
}
```

Data Transfer - Single Buffer

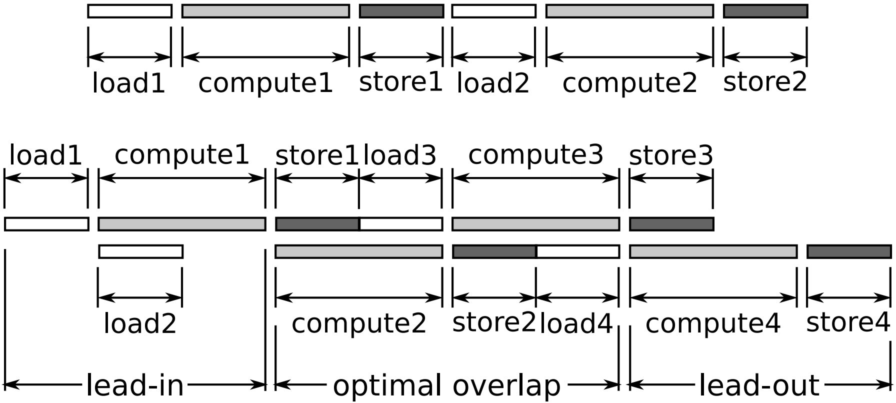
```
ii = in.get();
oo = out.get();

for(int i=0; i<iterations; i++) {

  spe_ppe_get_c(in.get(), cd->inbuf1+(SPE_Rank()+i*SPE_Size())*slicesize*sizeof(float),
    slicesize_padded*sizeof(float));

  harris_simd(ii, oo, cd->slice_dimx, cd->slice_dimy, 0, PADY, buf1.get(), buf2.get(), buf3.get());

  spe_ppe_put_c(cd->outbuf1+(SPE_Rank()+i*SPE_Size())*slicesize*sizeof(float) +
    (cd->slice_dimx*PADY)*sizeof(float), oo, slicesize*sizeof(float));
}
```

load

calc

store

Data Transfer



Data Transfer - Double Buffering

```
spe_ppe_get_async_c(in1.get(), cd->inbuf1+SPE_Rank()*slicesize*sizeof(float), slicesize_padded*sizeof(float), 9);
```

lead-in

```
for(int i=0; i<iterations; i++) {
  if(i%2 == 0) {
    spe_ppe_get_async_c(in2.get(), cd->inbuf1+(SPE_Rank()+(i+1)*SPE_Size())*slicesize*sizeof(float),
      slicesize_padded*sizeof(float), 10);

    dma_synchronize_c(9); dma_synchronize_c(11);
    ii = in1.get(); oo = out1.get();
  } else {
    spe_ppe_get_async_c(in1.get(), cd->inbuf1+(SPE_Rank()+(i+1)*SPE_Size())*slicesize*sizeof(float),
      slicesize_padded*sizeof(float), 9);

    dma_synchronize_c(10); dma_synchronize_c(12);
    ii = in2.get(); oo = out2.get();
  }
```

load, sync

```
  harris_simd(ii, oo, cd->slice_dimx, cd->slice_dimy, 0, PADY, buf1.get(), buf2.get(), buf3.get());
```

calc

```
  if(i%2 == 0) {
    spe_ppe_put_async_c(cd->outbuf1+(SPE_Rank()+i*SPE_Size())* slicesize*sizeof(float) +
      (cd->slice_dimx*PADY)*sizeof(float),  out1.get(), slicesize*sizeof(float), 11);
  } else {
    spe_ppe_put_async_c(cd->outbuf1+(SPE_Rank()+i*SPE_Size())* slicesize*sizeof(float) +
      (cd->slice_dimx*PADY)*sizeof(float), out2.get(), slicesize*sizeof(float), 12);
  }
}
```

store

```
spe_ppe_put_c(cd->outbuf1 + (SPE_Rank()+((iterations-1)*SPE_Size())) * slicesize*sizeof(float) +
  (cd->slice_dimx*PADY)*sizeof(float), oo, slicesize*sizeof(float));
```

lead-out

Double Buffering - Operations - Input Segment

- Start loading first segment (lead-in)

  `operator =()`

- Start loading next segment

  `operator ++(int)`

- Wait for segment to be ready for computation

  `operator *()`

- Signal that computation on current segment is finished

  `operator ++(int)`

- Check if end of data is reached

  `operator ==()`

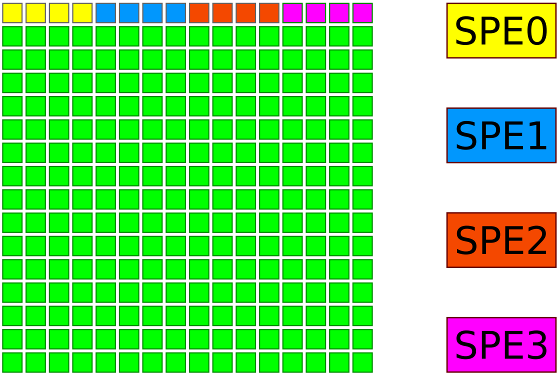
Double Buffered Segmented Input Iterator

```
template<typename T> struct remote_segmented_input_iterator
{
  // allocate required buffers
  remote_segmented_input_iterator(...) {}

  // start loading first buffer
  void operator= (const addr64 & base_address_) { }

  // wait for current segment to arrive and return pointer to it
  T* operator *() {}

  // start loading new data and increment current segment
  inline void operator++(int) {}

  // check if iterator has reached a position
  bool operator ==(const addr64 & b) const {}
};
```

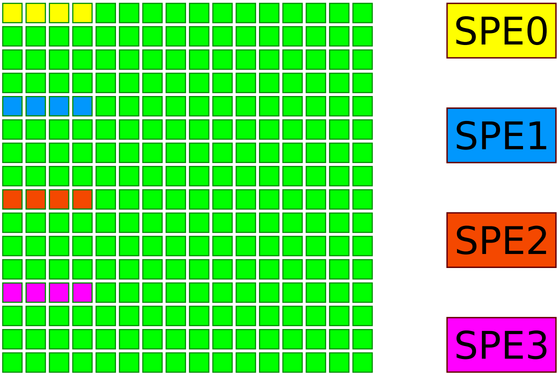
Double Buffered Segmented Iterator Example

```
remote_segmented_input_iterator<float> it(depth,
  ssize, slicer(ssize));
remote_segmented_output_iterator<float> ot(depth,
  ssize, slicer(ssize));

for(it = input, ot = output; /* lead-in */
    it!=input+overall_size; /* check end */
    it++, ot++) // load next, store current
{
  float * in = *it; float * out = *ot; // synchronize
  harris_simd(in, out, cd->slice_dimx, cd->slice_dimy,
   0, PADY, buf1.get(), buf2.get(), buf3.get());
}
```

Double Buffered Segmented Iterator - Slicer



Double Buffered Segmented Iterator - Slicer



Multi-Buffered Segmented Iterator - Features

- remote_vector<T> for more expressive code:

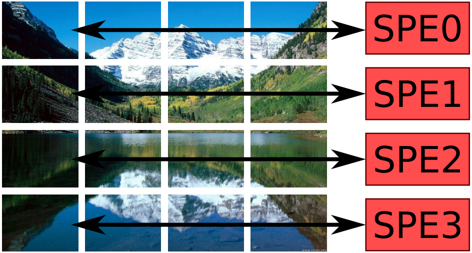
```
// PPE:
std::vector<float> v(1024*1024); kernel(v);
// SPE:
kernel(remote_vector<float> v) {
  remote_segmented_input_iterator<T> it(depth, ssize, slicer(ssize));
  for(it = v.begin(); it!=v.end(); it++) {
    float * in = *it;
    /* computation */
  }
}
```

- Read, write- and read-write Iterators with minimum buffer depth of 3
- Various slicers

2D Multi-Buffered Segmented Iterator

- Native 2D data transfer support through DMA lists
- Difference to regular iterator:
  - Slice size is 2D
  - Supports `remote_vector_2D`
  - Slicer takes 2D arguments:

    `slicer_2D(size_2d_t vector_dim, size_2d_t slice_dim);`
- Ideal for image processing:



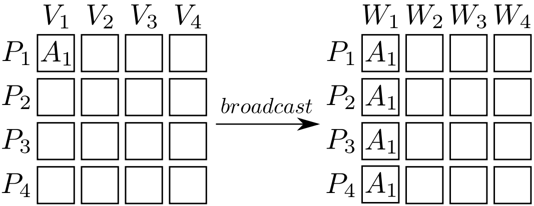
High-Level Inter-SPE Communication: MPI

- Interprocess communication by message passing, SPEs send and receive message
- API specification, used in high performance computing

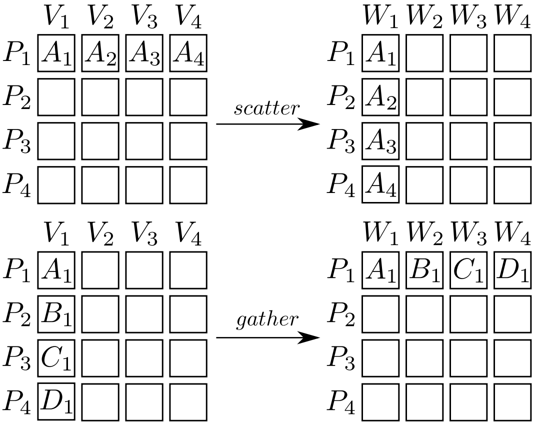
Features:

- Virtual topology of processes
- Synchronization
- Point to point communication
- Collective communication

MPI Collectives - Broadcast



MPI Collectives - Scatter and Gather



MPI Collectives - Reduce and All to All

$$
\begin{array}{c|c|c|c|c|}
 & V_1 & V_2 & V_3 & V_4 \\
P_1 & A_1 & & & \\
P_2 & B_1 & & & \\
P_3 & C_1 & & & \\
P_4 & D_1 & & &
\end{array}
\xrightarrow{\;reduce\;}
\begin{array}{c|c|c|c|c|}
 & W_1 & W_2 & W_3 & W_4 \\
P_1 & X & & & \\
P_2 & & & & \\
P_3 & & & & \\
P_4 & & & &
\end{array}
$$

$$X = Op(A_1, Op(B_1, Op(C_1, D_1)))$$

$$
\begin{array}{c|c|c|c|c|}
 & V_1 & V_2 & V_3 & V_4 \\
P_1 & A_1 & A_2 & A_3 & A_4 \\
P_2 & B_1 & B_2 & B_3 & B_4 \\
P_3 & C_1 & C_2 & C_3 & C_4 \\
P_4 & D_1 & D_2 & D_3 & D_4
\end{array}
\xrightarrow{\;alltoall\;}
\begin{array}{c|c|c|c|c|}
 & W_1 & W_2 & W_3 & W_4 \\
P_1 & A_1 & B_1 & C_1 & D_1 \\
P_2 & A_2 & B_2 & C_2 & D_2 \\
P_3 & A_3 & B_3 & C_3 & D_3 \\
P_4 & A_4 & B_4 & C_4 & D_4
\end{array}
$$

MPI Interface - Example

```
communicator world;

if (world.rank() == 0)
{
  char s1[] = "Hello";
  world.send(1, 0, s1, sizeof(s1));
  char s2[6];
  world.recv(1, 1, s2, sizeof(s2));
}
else if (world.rank() == 1)
{
  char s1[6];
  world.recv(0, 0, s1, sizeof(s1));
  char s2[] = "world";
  world.send(0, 1, s2, sizeof(s2));
}
// Hello world from SPE 0, Hello world from SPE 1
```

MPI Interface - Communicator

```
class communicator
{
  void barrier();

  template <typename T> void send(int dst, int tag, const T& value);
  template <typename T> void send(int dst, int tag, const T* values, int n);
  template <typename T> request isend(int dst, int tag, const T& value);
  ...
  template <typename T> status recv(int source, int tag, T& value);
  template <typename T> status recv(int source, int tag, T* values, int n);
  template <typename T> request irecv(int source, int tag, T& value);
  ...
  communicator include(uint16_t first, uint16_t last);
  communicator exclude(uint16_t first, uint16_t last);
  friend bool operator== (const communicator& c1, const communicator& c2);
};
```

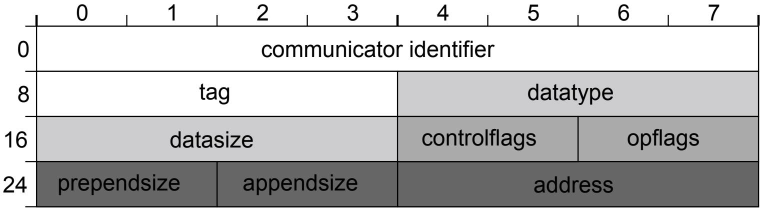
MPI Interface - Request and Status

```cpp
// represents current request
class request
{
  request() {};
  status wait();
  boost::optional<status> test();
};

// represents status of a request
class status
{
  int32_t source() const;
  int32_t tag() const;
  int32_t error() const;
};
```

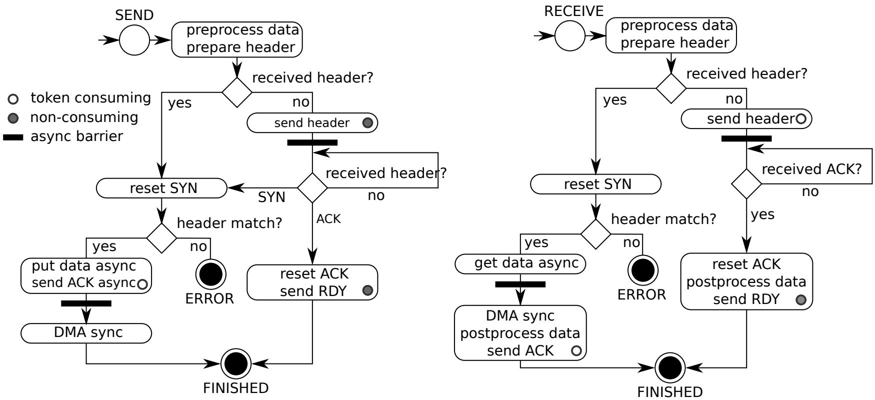
MPI Interface - Collectives Interface

```
template<typename T, typename Op>
void reduce(const communicator & comm, const T & in,
  T & out, Op op, int root);

template<typename T, typename Op>
void reduce(const communicator & comm, const T & in,
  Op op, int root);

template<typename T, typename Op>
void reduce(const communicator & comm, const T * in,
  int n, T * out, Op op, int root);

template<typename T, typename Op>
void reduce(const communicator & comm, const T * in,
  int n, Op op, int root);
```

MPI Header


0
1
2
3
4
5
6
7
0
communicator identifier
8
tag
datatype
16
datasize
controlflags
opflags
24
prependsize
appendsize
address

MPI Protocol



MPI Types

We don't do Boost.Serialization but

- you may register your POD type:

```
struct gps_position { /* POD */ };
namespace cbe_mpi
{
  CBE_MPI_USER_POD_DATATYPE(gps_position);
}
```

- or you may specialize send/receive methods:

```
template <typename T>
request isend(cbe_mpi::communicator & comm, int dst,
 int tag, T data, int n);

template <typename T>
request irecv(cbe_mpi::communicator & comm, int src,
 int tag, T data, int n);
```

Registering POD Types

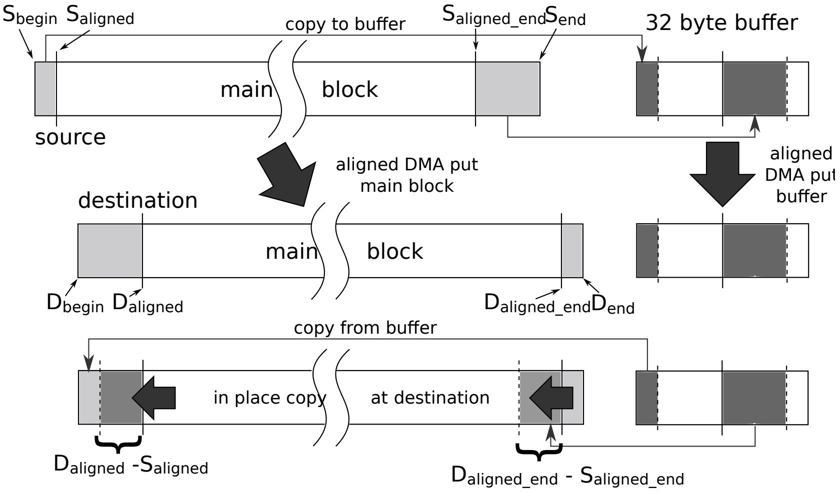
How we identify your type:

```
template<typename T>
struct cbe_mpi_user_pod_type_id { static void get() {} };

#define CBE_MPI_USER_POD_DATATYPE(CppType) \
template<> \
struct is_mpi_datatype< CppType > \
: boost::mpl::bool_<true> {}; \
      \
inline int get_mpi_datatype(const CppType &) \
{ \
  return 0x80000000 | \
    (int)(&cbe_mpi_user_pod_type_id< CppType >::get); \
}
```

Sending std::vector

```cpp
template <typename T>
request isend(cbe_mpi::communicator com,
              int dest, int tag, const std::vector<T> * values, int)
{
  int vectorsize = values->size();
  com.send(dest, tag, &vectorsize, 1);
  return com.isend(dest, tag, &(*values)[0], vectorsize);
}

template <typename T>
request irecv(cbe_mpi::communicator com,
              int source, int tag, std::vector<T> * values, int)
{
  int vectorsize;
  com.recv(source, tag, &vectorsize, 1);
  values->resize(vectorsize);
  return com.irecv(source, tag, &(*values)[0], vectorsize);
}
```

MPI - Sending Unaligned Data



Conclusion

- Build process can be simplified with CMake
- Boilerplate code can be simplified with the help of Boost (e.g. PP)
- Ambiguity of functions or macros in different compilation units can be exploited

- Optimal Boost solutions have to be adapted to fit embedded architecture
- Sweet spot between generic code and efficiency must be found

- Difficult low-level code can be wrapped nicely in C++ Concepts
- C++ Concepts can be even more powerful on special purpose hardware

Thank you for you kind attention.