Charm++ on the Cell Processor

David Kunzman, Gengbin Zheng,
Eric Bohm, Laxmikant V. Kale

Motivation

- Cell Processor (CBEA) is powerful (peak-flops)
  - Allow Charm++ applications utilize the Cell processor
- Cell Processor has a difficult architecture
  - Programmer specifically programs DMAs, local store management, and so on (diverts programmers attention from problem at hand)
  - Cell specific code interleaved in with application code
- Use the flexibility and abstracting abilities of the Charm++ programming model to help the programmer

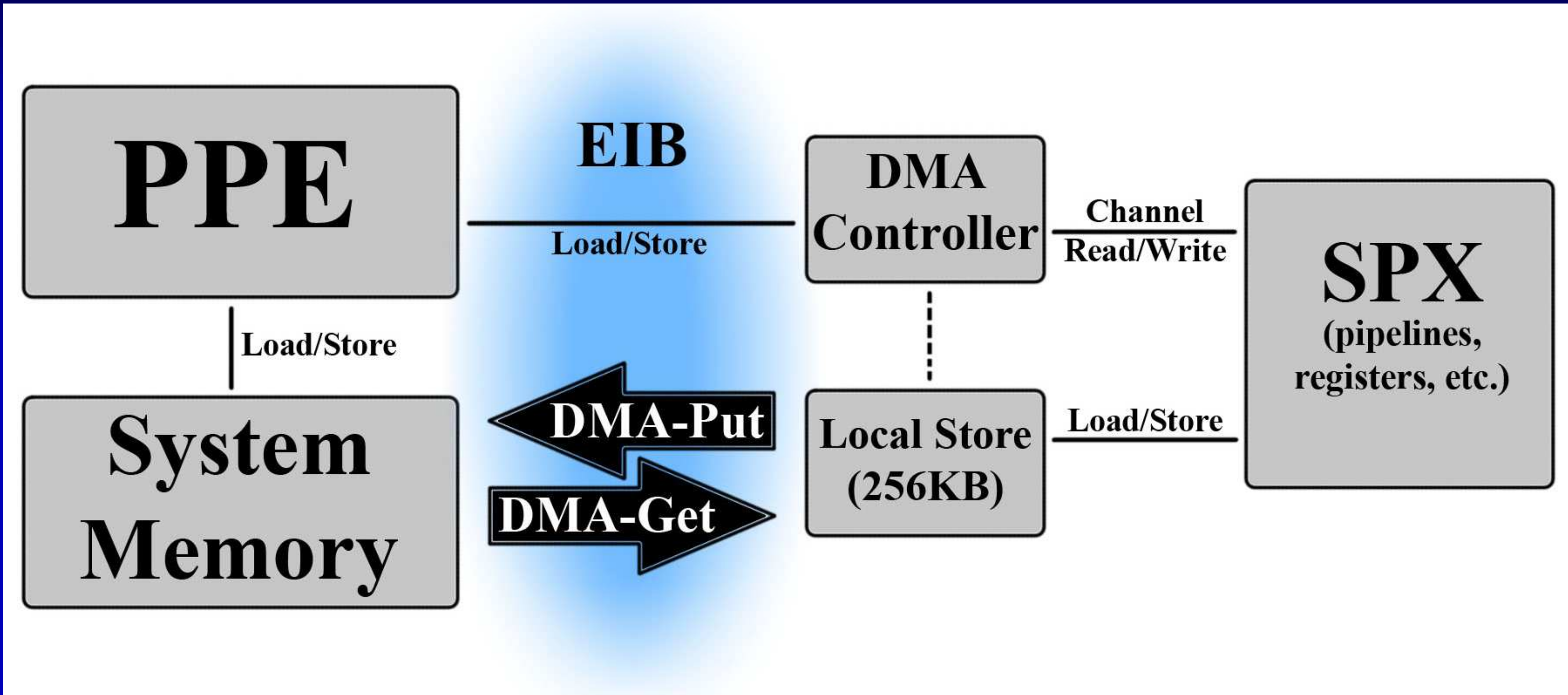
Overview of Talk

- Quick introduction to the Cell Processor
- Quick introduction to Charm++
- Affinity of Charm++ to the Cell processor
- Adaptation of Charm++ and Charm++ related tools

Cell Processor

- Power Processor Element (PPE) (x1)
  - Access to system memory
  - 2-way SMT
- Synergistic Processor Element (SPE) (x8)
  - No direct access to system memory
  - Local Store (LS): 256KB
  - DMA transactions to move data between system memory and LS

Cell Processor



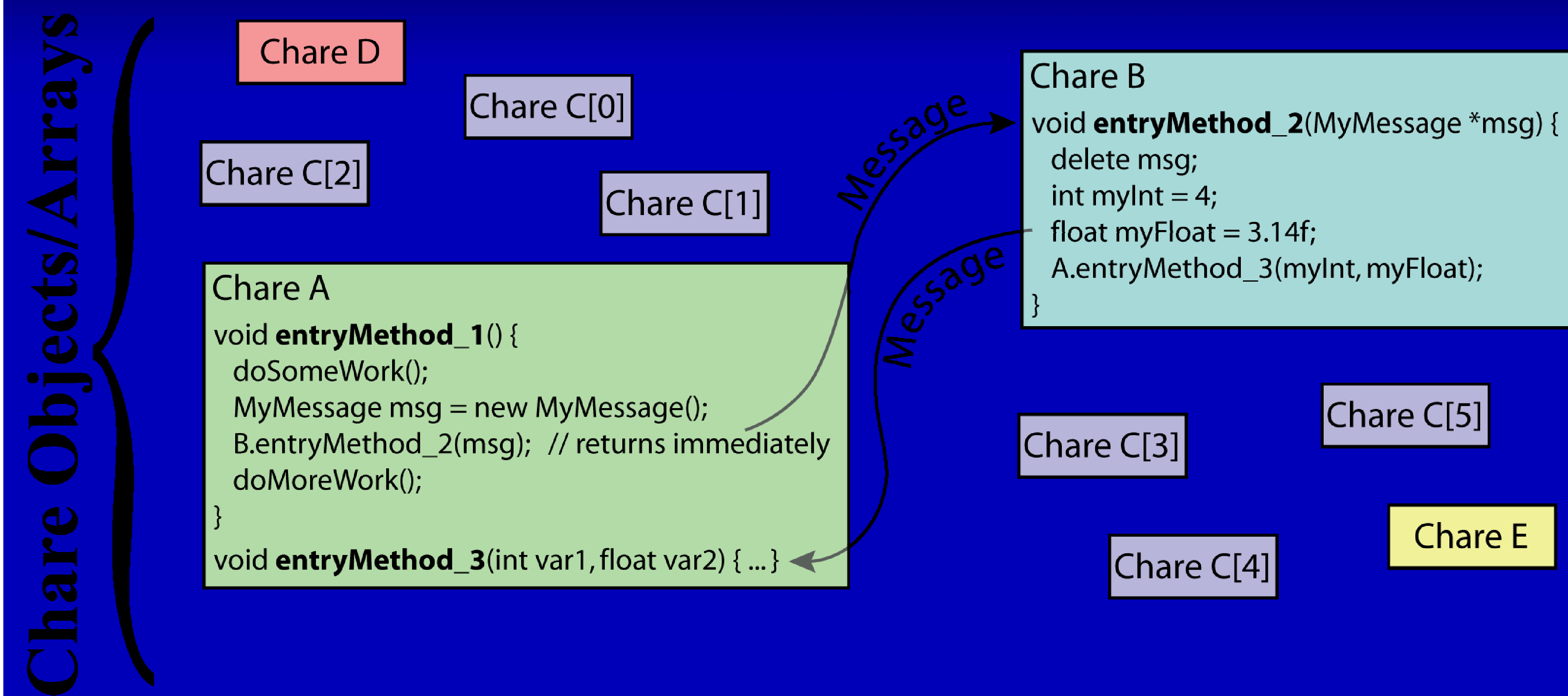
Charm++

- Object-Oriented, Message-Driven Parallel Programming Paradigm
  - Application broken up into objects called ***chares***
  - Chares communicate using asynchronous messages
  - Chares have special member functions called ***entry methods*** that receive messages
- Programmer doesn't worry about processors/interconnect/etc. when programming
- HPC Applications: Molecular Dynamics (NAMD), Cosmology (Changa), Rocket Simulation (Rocstar), etc.

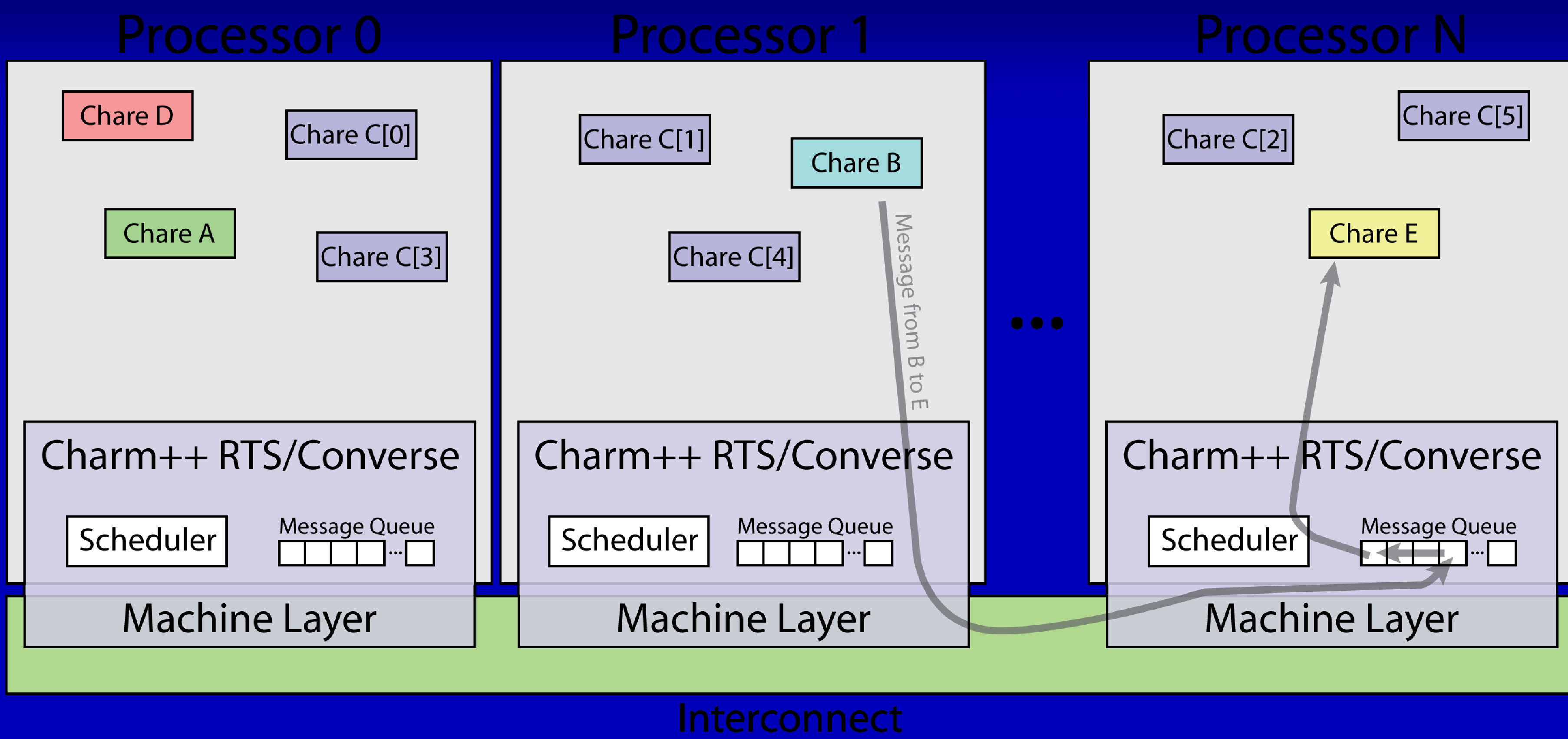
ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

PARALLEL PROGRAMMING LAB

User's View of Charm++



System View of Charm++

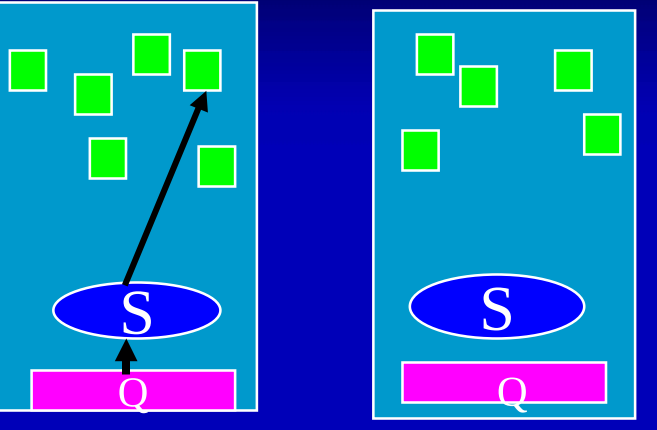


Observations

- Chares tend to be small
  - Increase concurrency in the application (many objects to spread across many processors)
- Chares tend to be self-contained
  - Their entry methods access data within the chare object itself and/or the message
- Via Pack-UnPack (PUP) routines, chares are migratable between processing elements

Why Charm++ & Cell?

- Data Encapsulation / Locality
  - Each message associated with…
    - Code : Entry Method
    - Data : Message & Chare Data
  - Entry methods tend to access data local to chare and message



- Virtualization (many chares per processor)
  - Provides opportunity to overlap SPE computation with DMA transactions
  - Helps ensure there is always useful work to do
- Message Queue Peek-Ahead / Predictability
  - Peek-ahead in message queue to determine future work
  - Fetch code and data before execution of entry method

Roadmap

(completed)

Phase 1
Charm++ Runtime System on PPE
(trivial)

(in progress)

Development of Offload API
Creation of the Offload API to allow arbitrary "chunks" of code to execute on the SPEs
(i.e. - allow Charm++ Runtime System to "offload" entry methods to SPEs)

(in progress)

Phase 2
Allow Charm++ applications (with modification) to take advantage of SPEs on multiple Cell chips.
(e.g. - 2D Jacobi program written in Charm++ has already been run across the 4 chips we have access to: 4 PPEs + 32 SPEs).

(in Progress)

Projections
Collection of performance data which can be visuallized using Projections.

(future work)

Phase 3
Modification of Charm++ tools (charmxi and/or charmc) so little to no code modification to the original Charm++ application is needed to allow it to run on Cell.

Development of Offload API

- Goal: Offload chunks of computation, called "Work Requests" onto the SPEs
- Design guided by needs of the Charm++ runtime system / programming model
  - However, independent of Charm++: Sequential C/C++ programs can use Offload API to utilize SPEs
- More Info:
  - In Papers section of http://charm.cs.uiuc.edu
    - Paper : 06-16 - "Charm++ on the Cell Processor"
    - Paper : 06-14 - "Charm++, Offload API, and the Cell Processor"

Basic Idea of Offload API

- User writes functions that execute on the SPE
- Each function takes input and output buffers
- User code handles…
  - Making a Work Request
    - Indicates which function to execute
    - Indicates where to get/put data in memory
- Offload API handles… (everything else)
  - Issuing Work Request to a particular SPE
  - Transferring input/output data (issuing DMA commands, managing local store memory)
  - Scheduling of Work Request execution

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

PARALLEL PROGRAMMING LAB

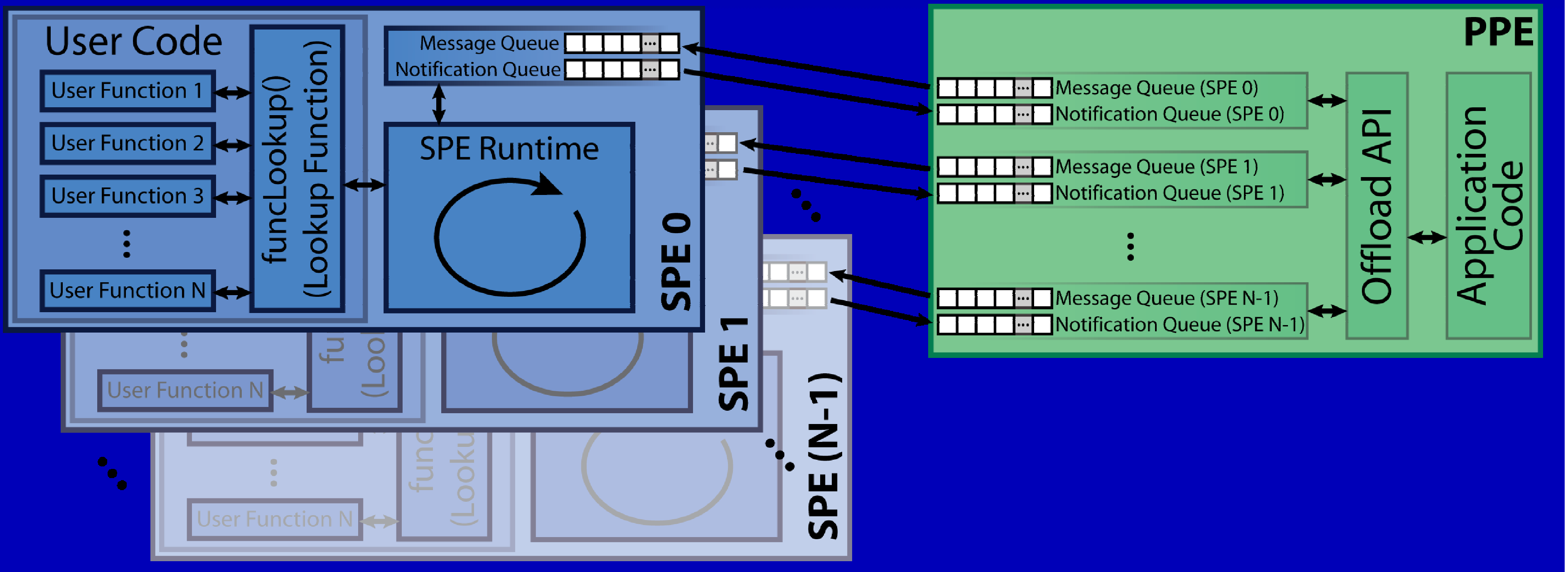
Offload API Code Example

```
///// hello.cpp (PPE Only) ///////////////////////
#include <stdio.h>
#include <string.h>
#include <spert_ppu.h> // Offload API Header
#include "hello_shared.h"
#define NUM_WORK_REQUESTS 10

int main(int argc, char* argv[]) {
  WRHandle wrHandle[NUM_WORK_REQUESTS];
  char msg[] __attribute__((aligned(128))) = { "Hello" };
  int msgLen = ROUNDUP_16(strlen(msg));

  InitOffloadAPI();

  // Send some work requests
  for (int i = 0; i < NUM_WORK_REQUESTS; i++)
    wrHandle[i] = sendWorkRequest(FUNC_SAYHI,
                                  NULL, 0,
                                  msg, msgLen,
                                  NULL, 0
                                 );

  // Wait for the work requests to finish
  for (int i = 0; i < NUM_WORK_REQUESTS; i++)
    waitForWRHandle(wrHandle[i]);

  CloseOffloadAPI();
  return EXIT_SUCCESS;
}
```

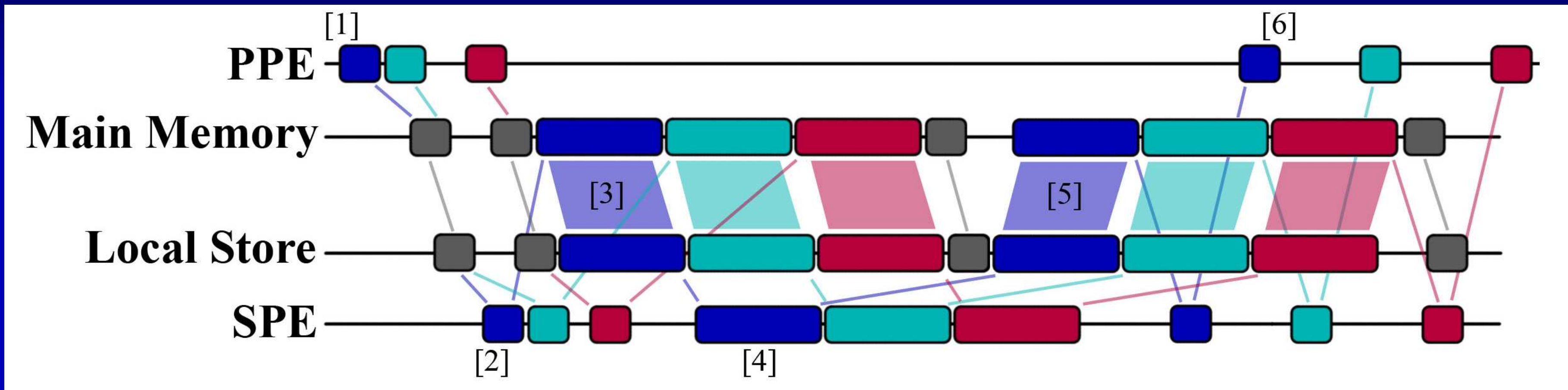
```
///// hello_spe.cpp (SPE Only) ////////////////
#include <stdio.h>
#include "spert.h" // SPE Runtime Header
#include "hello_shared.h"

inline void sayHi(char* msg) {
  printf("\"%s\" from SPE %d...\n",
         msg, (int)getSPEID());
}

#ifdef __cplusplus
extern "C"
#endif
void funcLookup(int funcIndex,
    void* readWritePtr, int readWriteLen,
    void* readOnlyPtr, int readOnlyLen,
    void* writeOnlyPtr, int writeOnlyLen,
    DMAListEntry* dmaList) {
  switch (funcIndex) {
    case SPE_FUNC_INDEX_INIT: break;
    case SPE_FUNC_INDEX_CLOSE: break;
    case FUNC_SAYHI:
      sayHi((char*)readOnlyPtr);
      break;
    default: // should never occur
      printf("ERROR :: Invalid funcIndex (%d)\n",
             funcIndex);
      break;
  }
}
```

```
///// Output //////
"Hello" from SPE 0...
"Hello" from SPE 7...
"Hello" from SPE 4...
"Hello" from SPE 5...
"Hello" from SPE 6...
"Hello" from SPE 2...
"Hello" from SPE 3...
"Hello" from SPE 0...
"Hello" from SPE 1...
"Hello" from SPE 1...
```

Structure of Offload API



SPE Runtime



- [1] : PPE Sends Work Request
- [2] : SPE Receives Work Request (through Work Request List)
- [3] : DMA-Get used to retrieve input data from system memory
- [4] : Work Request is executed
- [5] : DMA-Put used to place output data into system memory
- [6] : SPE notifies PPE of Work Request Completion

(NOTE : Not to Scale)

Phase 2

- Execution of Charm++ applications on Cell (with some modification to application code)
  - Allows…
    - Charm++ applications to take advantage of SPEs
    - Charm++ applications to run across multiple Cell chips
  - However…
    - Requires user code to explicitly issue Work Requests using Offload API

Charm++ Code Example

```
///// hello.C (PPE Only) /////////////////////
#include "hello_shared.h"

class Main : public CBase_Main {

 public:
  Main(CkArgMsg* m) {
    // ...
    CProxy_Hello arr = CProxy_Hello::ckNew(nElements);
    arr[0].SayHi(17);
  };

  void done(void) {
    CkPrintf("All done\n");
    CkExit();
  };
};

class Hello : public CBase_Hello {

 public:

  void SayHi(int hiNo) {
    char buf[16] __attribute__((aligned(16)));
    sprintf(buf, "%d", thisIndex);
    sendWorkRequest(FUNC_SAYHI,
                    NULL, 0,          // RW
                    buf, strlen(buf)+1, // RO
                    NULL, 0,          // WO
                    CthSelf()
                   );
    CthSuspend();

    if (thisIndex < nElements-1)
      thisProxy[thisIndex+1].SayHi(hiNo+1);
    else
      mainProxy.done();
  }
};
```

```
///// hello_spe.cpp (SPE Only) ///////////////
#include "spert.h"
#include "hello_shared.h"

#ifdef __cplusplus
extern "C"
#endif
void funcLookup(int funcIndex,
                void* readWritePtr, int readWriteLen,
                void* readOnlyPtr, int readOnlyLen,
                void* writeOnlyPtr, int writeOnlyLen,
                DMAListEntry* dmaList
               ) {

  switch (funcIndex) {
    case FUNC_SAYHI: sayHi((char*)readWritePtr,
          (char*)readOnlyPtr); break;
    default:
      printf("!!! WARNING !!! Invalid funcIndex (%d)\n",
          funcIndex);
      break;
  }
}

void sayHi(char* readWritePtr, char* readOnlyPtr) {
  printf("\"Hi\"... \"%s\"\n", readOnlyPtr);
}
```

```
///// hello.ci //////
mainmodule hello {
  readonly CProxy_Main mainProxy;
  readonly int nElements;

  mainchare Main {
    entry Main(CkArgMsg *m);
    entry void done(void);
  };

  array [1D] Hello {
    entry Hello(void);
    entry [threaded] void SayHi(int hiNo);
  };
};
```

///// Output //////
"Hi"… "0"
"Hi"… "1"
"Hi"… "2"
"Hi"… "3"

…
(through nElements lines)

Projections

- Projections is a performance visualization tool used to access the performance of Charm++ applications
  - Allows user to visually see how the processing elements are being utilized through various graphs/tools

Projections - Timeline

Phase 3 (future work)

- Modification of Charm++ runtime system and tools
  - Charmxi
    - Generate SPE code from user's code (from entry method code)
    - Generate funcLookup() function for the user
  - Runtime implicitly generates work requests when pulling entries off the message queue
    - Object data, code for entry method, and message passed as part of work request
    - Removes overhead of having to enter user code
    - User does not have to issue Work Requests directly from application code

Summary - Roadmap

(completed)

Phase 1
Charm++ Runtime System on PPE
(trivial)

(in progress)

Development of Offload API
Creation of the Offload API to allow arbitrary "chunks" of code to execute on the SPEs
(i.e. - allow Charm++ Runtime System to "offload" entry methods to SPEs)

(in progress)

Phase 2
Allow Charm++ applications (with modification) to take advantage of SPEs on multiple Cell chips.
(e.g. - 2D Jacobi program written in Charm++ has already been run across the 4 chips we have access to: 4 PPEs + 32 SPEs).

(in Progress)

Projections
Collection of performance data which can be visuallized using Projections.

(future work)

Phase 3
Modification of Charm++ tools (charmxi and/or charmc) so little to no code modification to the original Charm++ application is needed to allow it to run on Cell.

Acknowledgements

- Thanks to…
  - Everyone at IBM that has helped with this work (especially Hema Reddy and Bob Szabo)
  - NCSA for allowing us to use their Cell Blades

Questions?

More Involved Example: 2D Jacobi

- 2D Jacobi written using Charm++ with required modifications for Cell (same as 5-point stencil)
  - PPE handles communication via Charm++ model
  - Actual calculation offloaded to SPEs via Work Requests
  - Code in Charm++ Distribution: [charmDir]/examples/charm++/cell/jacobi/