Charm++ Simplifies Programming for the Cell Processor

David Kunzman, Gengbin Zheng, Eric Bohm, Jim Phillips, Laxmikant V. Kalé

PARALLEL PROGRAMMING LAB

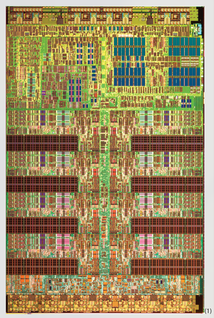
Introduction

The Cell processor, jointly developed by IBM, Sony, and Toshiba, has enormous computational processing power. It also deviates from traditional processor design. We at the Parallel Programming Lab, believe that several features of the Charm++ programming model make it a good fit for the Cell Broadband Engine Architecture (CBEA or "Cell"), including data encapsulation, virtualization, peek-ahead ability in the message queues, and portability.

By porting the Charm++ Runtime System to Cell-based platforms, we hope to bring the rich set of features available to Charm++ applications to the Cell. Towards this end, we have begun porting the Charm++ Runtime System to the Cell. Here, we present the current progress and future directions of this effort.

The Cell Processor

The Cell processor deviates from typical processor design. The Cell has nine cores. The "main" core, called the Power Processor Element (PPE), can be thought of as a standard 2-way SMT processing core. The other eight cores are specialized to do large amounts of computation quickly.



Power Processor Element (PPE)

- Similar to standard PowerPC core
- 2-way SMT
- Can access system memory using loads and stores

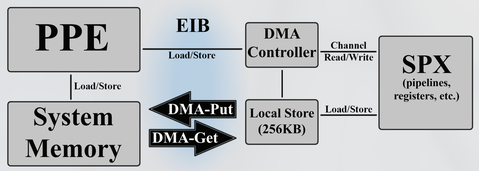
Element Interconnect Bus (EIB)

- Connects all elements in the Cell (PPE, SPEs, System Memory, I/O)
- Transfer rate: 96 bytes/cycle (200 GB/s best-case)
- Two Cell chips may connect via EIB and I/O ports

Synergistic Processor Element (SPE)

- Same clock speed as PPE
- Performance : 25.6 GFlop/s peak (single-precision: 8 flops / cycle using FMA x 3.2 GHz)
- SPE can only access Local Store (LS) : 256 KB, Holds everything (data, code, stack), 6-cycle Latency
- Data moved between System Memory and LS using DMA transactions
  - DMA transactions are cache coherent
  - PPE can also issue DMA transactions
- Two in-order pipelines (maximum of two instructions per clock)

Communication Within the Cell



Offload API

The Offload API has been developed by the Parallel Programming Lab (PPL). Its purpose is to enable the Charm++ Runtime System, and thus Charm++ applications, to utilize the SPEs on the Cell processor. While this is the main purpose of the Offload API, it has also been developed to be independent of Charm++ and can be used by any C/C++ application. The Offload API is publicly available as part of the Charm++ distribution. This includes the source code, along with some simple example programs (both Charm++ and general C/C++ programs). Current development efforts are focused on enhancing the Offload API as well as decreasing the overhead produced by the Offload API itself.

Work Requests (WR)

- Well Defined Input and Output (buffers)
- Three types of buffers (aimed at reducing EIB traffic)
  - Read/Write: Data copied to/from Local Store before/after WR execution, respectively
  - Read-Only: Data only copied to LS before WR execution
  - Write-Only: Data only copied out of LS after WR execution
- Types of Work Requests
  - Scatter/Gather (using DMA Lists): many of each type of buffer
  - Standard: one of each buffer type, requires less setup in user code
- Self-Contained Code: no dependencies with other chunks of code or Work Requests that are executed concurrently, *i.e. one shouldn't issue two WRs that both write to the same memory buffer since the order-of-execution between concurrent WRs is not enforced*
- Work Request code is executed on the SPEs
- Migration of Input/Ouput buffers to/from LS is overlapped with useful computation on SPE (double buffering takes place at the granularity level of WRs)
- PPE notified of Work Request completion
  - User can specify a callback function that will be called by the Offload API
  - If no callback specified, Work Request Handle can be used to check if a Work Request has completed (non-blocking) or wait until a Work Request completes (blocking)

Work Request Groups

- A set of Work Requests logically grouped together
- Use code notified of Work Request Group completion either via callback function specified by user or by polling/blocking on the Work Request Group Handle
- Less overhead (less callbacks/notifications)

SPE Runtime

- PPE queues Work Requests to SPE and periodically checks for completion, SPE Runtime responsible for:
  - Issuing DMA commands to move data buffer between main storage and SPE's local store
  - Executing Work Requests
  - Managing Local Store
  - Notifying PPE when Output Buffers have been placed in system memory (Work Request completed)
- SPE Runtime runs asynchronously to PPE code
- Requires lookup function which maps function indexes to actual function pointers (user code)

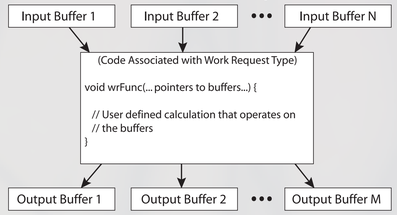
Current Status

- Offload API usable *(however, still actively under development)*
- Distributed as part of Charm++
  - Includes source code
  - Includes simple example programs

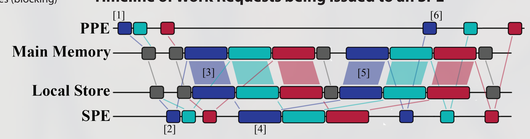
Core Idea

The overall computation performed by an application is broken up into chunks of smaller computations called *Work Requests*. PPE application code issues these work requests to the Offload API. The Offload API then passes the work requests to the SPEs. When a work request has finished executing and its output has been moved into system memory, the PPE code is notified. Each individual work request is executed in its entirety on a single SPE. The idea is to have many work requests pending at a time. First, this helps ensure that all SPEs have work to perform. Second, it allows the Offload API to overlap the movement of one work request's data with the computation of another work request.

Structure of a Work Request



Timeline of Work Requests being Issued to an SPE



This figure shows the flow of three work requests being passed to a single SPE. Following the blue work request, code on the PPE makes a work request [1]. The SPE then initiates a DMA-Get [2] to retrieve the needed data. Once the data has arrived in the local store [3], the work request is executed on the SPE [4]. After completion, the output data is transferred into system memory [5] and the PPE is notified that the work request has finished [6]. *(Not to Scale: [4] should be greater than [3] + [5])*

Future Work

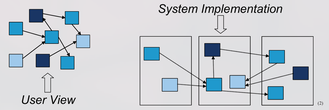
- Strided Work Request Type
  - Multiple calls to the same Work Request function without PPE intervention (to reduce PPE overhead)
- Each input/output buffer is divided into chunks
  - Sizes of the chunks (per buffer)
  - Strides (pointer offsets) between chunks (per buffer)
    - Zero used to *share* the same buffer between calls
- Less total overhead
  - PPE only involved at very beginning and very end
  - LS memory reused for all invocations of Work Request function
- Additional Work Request types based on usage patterns in applications and/or the Charm++ RTS
- Performance testing on actual Cell hardware *(has already been started)*
- Reduce the overhead caused by the SPE Runtime
  - Memory: Reduce the code size so more of the local store is available to the application
  - Time: Reduce the amount of time spent executing SPE Runtime code to process the Work Requests
- Add ability to move code to/from the Local Stores

For more information, please visit: http://charm.cs.uiuc.edu

Charm++

Core Idea

Charm++ has traditionally been used to create high performance computing (HPC) applications. In the Charm++ programming model, the program is broken down into objects called "chares." Each chare is responsible for a portion of the overall computation. The chares communicate with each other via asynchronous method invocation (messages). The programmer writes the application in terms of these chares (instead of processors). Typically, the number of chares is much greater than the number of processors. The Charm++ Runtime System takes care of mapping the objects to processors and routing messages to the correct processors.



Each chare has one or more *entry methods* defined. When a chare sends a message to another chare, it must also specify the entry method that will be invoked on the receiving chare. Once the receiving chare receives the message, then entry method is executed with the message being passed to it as a parameter. The entry method, a member function of the chare object, can then do some computation, including sending more messages to other chares.

Suitability to the Cell Processor

- **Data Encapsulation**: Data needed by entry methods is encapsulated within the chare object itself and within the message that the chare just received
- **Virtualization** (many chares per physical processor): With many chares on each processor, typically at least one chare has a message waiting for it and thus is ready to executed an entry method. While this entry method executes, the other chares on the processor wait for pending messages to arrive (overlap of computation and communication effectively hiding message latency).
- **Peek-Ahead in Message Queue**: As messages arrive at the processor, they are queued by the Charm++ RTS. The Charm++ RTS can then peek-ahead in this queue and preemptively push the data (and code) needed by a future entry method to the SPE's Local Store. When the SPE finishes the current entry method, it will have the data (and code) needed to immediately start executing the next entry method.

Charm++ on Cell

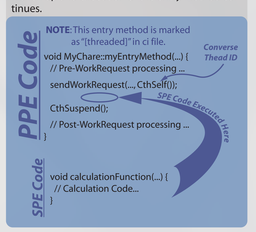
- Impractical to map Chares to SPEs (LS cannot hold many, leaving most on PPE)
- Instead, entry methods will be executed on the SPEs
  - Each entry method executed on SPE as a single work request
  - Chare data and message will be moved to SPE's Local Store prior to the entry method being executed (while the SPE is currently executing another entry method)
  - Only entry methods that are *safe* to execute on the SPEs will execute on them (i.e. - entry methods that do not randomly access global data, etc.).
  - *Non-safe* entry methods will execute on the PPE

Charm++ and the Offload API

The Charm++ Runtime System uses the Offload API to *offload* portions of the computation (entry methods) performed by Charm++ applications to the SPEs.

Current Status

Currently, for a Charm++ application to take advantage of the SPEs, some code changes are required. A Charm++ application must explicitly make calls to the Offload API. Entry methods that *offload* work onto the SPEs must be *threaded*. This causes the Charm++ Runtime System to create a light-weight Converse thread when executing the entry method. After the entry method issues a Work Request to the Offload API, the entry method then suspends. Once the work request has completed, the thread executing the entry method is woken up and execution of the entry method continues.



Future Goal

Eventually, each chare may have one or more entry methods that are *safe* and will be *offloaded* onto the SPEs. This will allow the user to write a Charm++ application just like they would for any other platform. During compilation of the program, the Charm++ tools will generate the necessary wrapper code for entry methods that can be safely executed on the SPEs (i.e. - entry methods that do not randomly access global data structures). The Charm++ Runtime System will then execute these *safe* entry methods on either the PPE or one of the SPEs as it sees fit.

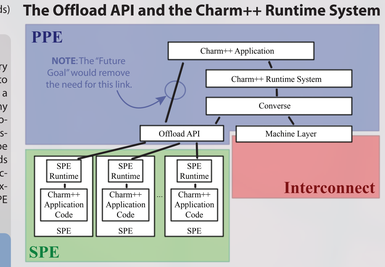
Charm++ Code

```
void MyChare::myEntryMethod(...) {
  // Calculation Code...
}
```

NOTE: Charmc and/or charmxi will generate the PPE stub code and SPE code needed by the Offload API as long as the "Calculation Code" is safe to execute on the SPEs. The generated code would hook directly into the Charm++ RTS, removing the need for Converse threads. However, determination of whether or not an entry method is "safe" may require, at least initially, the user to specify a keyword in the ci file (such as "[offloadable]" for example).

Chare Object Data: Read/Write Buffer
Incoming Message(s): Read-Only Buffer(s)
Generated Message(s): Write-Only Buffer(s)

The Offload API and the Charm++ Runtime System



Charm++ Application: User application code that has been compiled using the Charm++ tools.

Charm++ Runtime System: Contains support for Charm++ abstractions (chares, chare arrays, etc.)

Converse: An abstract message passing layer; contains functionality and/or interface to send point-to-point messages, broadcast messages, etc.

Machine Layer: Directly interacts with hardware providing functionality needed by both Converse and the Charm++ Runtime System

For more information, please visit: http://charm.cs.uiuc.edu

Simple Offload API Code Example ("Hello")

```
///// hello_shared.h (PPE + SPE) //////////////
#ifndef __HELLO_SHARED_H__
#define __HELLO_SHARED_H__

#define FUNC_SAYHI 1

#endif //__HELLO_SHARED_H__
```

```
///// Output //////////////////////////////////
$ ./hello
"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...
```

NOTE: *If sayHi() is moved to shared_hello.h, a simple define could be used to either call sendWorkRequest() if the code is being compiled on a Cell platform or call sayHi() directly from main() if not on a Cell platform.*

```
///// 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 requets 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: printf("ERROR :: Invalid funcIndex\n"); break;
  }
}
```

NOTE: *This function is used by the SPE Runtime to execute user code. It simply maps functions to function indexes along with the appropriate mapping of parameters (buffers).*

Charm++ Code Example ("Hello")

```
///// 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 void sayHello(int num);
  };
};
```

```
///// Output //////////////////////////////
$ charmrun +p4 ./hello
Running Hello on 4 processors for 10 elements
Hello from element 0 on processor 0 (num:3)
Hello from element 4 on processor 0 (num:3)
Hello from element 8 on processor 0 (num:3)
Hello from element 2 on processor 2 (num:3)
Hello from element 6 on processor 2 (num:3)
Hello from element 3 on processor 3 (num:3)
Hello from element 7 on processor 3 (num:3)
Hello from element 1 on processor 1 (num:3)
Hello from element 5 on processor 1 (num:3)
Hello from element 9 on processor 1 (num:3)
```

NOTE: *Header file (hello.h) not used for brevity.*

```
///// hello.C ////////////////////////////////////////////////////////////////////////////////////////////////////
#include <stdio.h>
#include "hello.decl.h"  // Generated by Charm++ tools

/*readonly*/ CProxy_Main mainProxy;
/*readonly*/ int nElements;

// The "Main" Chare
class Main : public CBase_Main {
  public:

  Main(CkArgMsg* m) {
    // Process command-line arguments
    nElements = 10;  // Default to 10 "Hello"s
    if ((m->argc) > 1) nElements = atoi(m->argv[1]);
    delete m;

    // Print info on the run for the user
    CkPrintf("Running Hello on %d processors for %d elements\n",
             CkNumPes(), nElements
            );

    // Create a global reference to the Main chare's proxy
    mainProxy = thisProxy;

    // Create the array of "Hello" chares and then broadcast
    //  a message to them instructing them to say hi
    CProxy_Hello arr = CProxy_Hello::ckNew(nElements);
    srand(0);
    arr.sayHello(rand() % 10);
  };
```

...continued in next column...

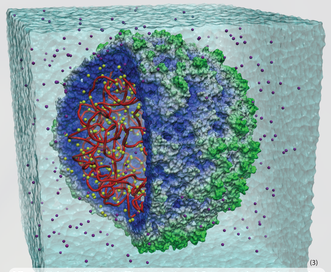
```
  void done(void) {
    static int helloFinishedCount = 0;

    // Increment the finished counter and check to see
    //  if all elements of the "Hello" array have completed
    helloFinishedCount++;
    if (helloFinishedCount >= nElements)
      CkExit();
  };
};

// A 1D Array of Chares
class Hello : public CBase_Hello {
  public:

  Hello() { }                          // General Constructor
  Hello(CkMigrateMessage *m) { }       // Migration Constructor
  void sayHello(int num) {

    // Say "Hello"
    CkPrintf("Hello from element %d on processor %d (num:%d)\n",
             thisIndex, CkMyPe(), num
            );

    // Send a message to the main chare indicating that
    //  this element has finished its "computation"
    mainProxy.done();
  }
};

#include "hello.def.h"  // Generated by Charm++ tools
```

NAnoscale Molecular Dynamics (NAMD)

NAMD is a Charm++ application, jointly developed by the Theoretical and Computation Biophysics Group and the Parallel Programming Lab, used to simulate bio-molecular structures. NAMD is quite popular, with over 17,000 users throughout the world. It is widely regarded as one of the fastest and most scalable codes for molecular dynamics (MD). In 2002, it won the Gorden Bell Award for scaling to 3,000 processors. Since then, NAMD has demonstrated good scaling up to 8,000 processors on Blue Gene/L.

Highlights

- 2002 Gordon Bell Award: Scaled to 3000 processors
- To date: Good scaling on Blue Gene/L up to 8,000 processors
- Over 17,000 registered users worldwide (including many DOE labs and supercomputing centers)
- Selected as one of the core applications for upcoming NFS petascale supercomputing project (Track 1)
- Used for first simulation of an entire lifeform at the atomic level (see image to the right)



Virus: Using NAMD, the Theoretical and Computational Biophysics Group, at the Beckman Institute (UIUC), were the first to simulate an entire life form at the atomic level. By using 256 Itanium2 processors in NCSA's Cobalt cluster for 9 days, they were able to simulate 10ns in the life of the Satellite Tobacco Mosaic Virus (pictured above). The simulation consisted of approximately 1 million atoms.

Problem Decomposition

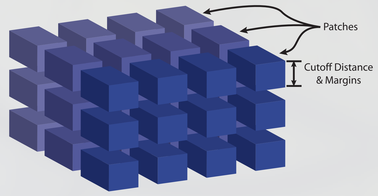
- 3D Space is broken down into a grid of 3D blocks (called *Patches*)
  - Size of Patch determined by cutoff distance (plus margin values: hydrogen bond length, etc.)
- Bonded and non-bonded forces are calculated using compute objects
  - Broken down by type (for example: non-bonded electrostatic, angle, etc.)
  - Broken down by data (for example: each pair of neighboring patches has an associated non-bonded electrostatic compute object)
- Integration of force data done by Patch Objects
- Long distance electrostatics done by using Particle Mesh Ewald (PME) calculation
- Periodically, atoms migrate between Patch Objects as necessary
- Typically, 1 femtosecond timesteps

NAMD on Cell

Initially, the non-bonded electrostatic force computation will be offloaded to the SPEs. This is our first step for several reasons:

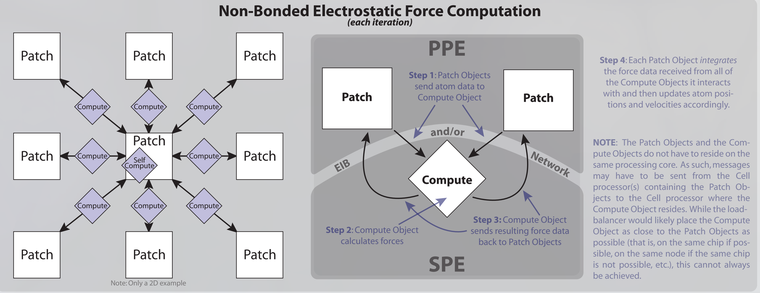
- Non-bonded forces represent approximately 80% of overall computation per iteration
- Clearly defined input (two input messages containing atom data)
- Clearly defined output (two output messages containing force data)
- Computes and Patches are clearly separated (interacting via messages/buffers)
- Charm++ Runtime System can freely migrate Computes and Patches independently

Once completed, other portions of the computation will be offloaded to the SPEs as needed.



Non-Bonded Electrostatic Force Computation

(each iteration)



NOTE: The Patch Objects and the Compute Objects do not have to reside on the same processing core. As such, messages may have to be sent from the Cell processor(s) containing the Patch Objects to the Cell processor where the Compute Object resides. While the load-balancer would likely place the Compute Object as close to the Patch Objects as possible (that is, on the same chip if possible, on the same node if the same chip is not possible, etc.), this cannot always be achieved.

For more information, please visit: http://www.ks.uiuc.edu/Research/namd

Conclusion

Offload API

- Currently usable
  - Part of Charm++ distribution
  - Includes source code and simple example programs
- Stand-alone interface (can be used independently of Charm++)
- Does:
  - Simplify programming
  - Add affinity to Charm++ programming model (enabling natural offloading of entry methods to SPEs)
  - Allow programmer to use SIMD instructions (and other SPE specific code such as SPE intrinsics, DMAs, etc.)
- Does not:
  - Hide all details from programmer (for example, data buffers still require proper alignment)
  - Allow programmer to be completely unaware of Cell architecture

Charm++

- Several aspects of the Charm++ model fit well with Cell processor's architecture
  - Data encapsulation:
    - Data: contained within chare and arriving message
    - Code: entry method
  - Virtualization (many chares per processor)
  - Peek-Ahead in Message Queue (Charm++ RTS *knows* what entry methods need to be executed in the near future; initiate DMAs prior to SPE *needing* data/code)
- Currently, requires some modification to Charm++ application code
- Example Charm++ applications which directly use the Offload API are included within the Charm++ distribution

Future Work / Directions

- Performance testing on actual hardware (just started this)
  - Performance of Offload API
    - Latency of Work Request
    - Overlap of Work Request execution and Work Request data movement
  - Performance of Charm++ applications
    - Overhead of Charm++ Runtime System (as increase by Offload API)
- Portability
  - Extend charmc and charmxi tools to auto-generate the needed Cell specific code for entry methods
  - Eventual goal: no modification to Charm++ applications when moving them between Cell-based and non-Cell-based platforms
- If appropriate, extend abstraction to GPUs and/or FPGAs
  - To begin with
    - Allow user to specify GPU and/or FPGA versions of the code
    - Charm++ Runtime System executes entry methods on GPU/FPGA if hardware is present
  - Eventual goal: Allow GPU and FPGAs to use the same framework as Offload API to *offload* entry methods

References / Related Papers

Charm++: L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, Parallel Programming using C++, pages 175–213. MIT Press, 1996.

Charm++ on Cell: D. Kunzman, G. Zheng, E. Bohm, L. V. Kale. "Charm++, Offload API, and the Cell Processor." In PMUP Workshop at PACT'06, Sept. 2006.

NAMD: J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with NAMD. Journal of Computational Chemistry, 26(16):1781–1802, 2005.

Images Sources:

(1) : Courtesy of International Business Machines Corporation. Unauthorized use not permitted.
(2) : From Parallel Programming Lab's (PPL) Website (http://charm.cs.uiuc.edu).
(3) : Courtesy of the Theoretical and Computational Biophysics Group, Beckman Institute, UIUC Website: http://www.ks.uiuc.edu

11/14/2006

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN