Charm++
Motivations and Basic Ideas

Laxmikant (Sanjay) Kale
http://charm.cs.illinois.edu
Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana Champaign
Challenges in Parallel Programming

• Applications are getting more sophisticated
  – Adaptive refinements
  – Multi-scale, multi-module, multi-physics
  – E.g. Load imbalance emerges as a huge problem for some apps
• Exacerbated by strong scaling needs from apps
• Future challenge: hardware variability
  – Static/dynamic
  – Heterogeneity: processor types, process variation, ..
  – Power/Temperature/Energy
  – Component failure
• To deal with these, we must seek
  – Not full automation
  – Not full burden on app-developers
  – But: a good division of labor between the system and app developers
What is Charm++?

• Charm++ is a generalized approach to writing parallel programs
  – An alternative to the likes of MPI, UPC, GA etc.
  – But not to sequential languages such as C, C++, and Fortran

• Represents:
  – The style of writing parallel programs
  – The runtime system
  – And the entire ecosystem that surrounds it

• Three design principles:
  – Overdecomposition, Migratability, Asynchrony
Overdecomposition

• Decompose the work units & data units into many more pieces than execution units
  – Cores/Nodes/..

• Not so hard: we do decomposition anyway
Migratability

• Allow these work and data units to be migratable at runtime
  – i.e. the programmer or runtime, can move them

• Consequences for the app-developer
  – Communication must now be addressed to logical units with global names, not to physical processors
  – But this is a good thing

• Consequences for RTS
  – Must keep track of where each unit is
  – Naming and location management
Asynchrony: Message-Driven Execution

• Now:
  – You have multiple units on each processor
  – They address each other via logical names

• Need for scheduling:
  – What sequence should the work units execute in?
  – One answer: let the programmer sequence them
    • Seen in current codes, e.g. some AMR frameworks
  – Message-driven execution:
    • Let the work-unit that happens to have data (“message”) available for it execute next
    • Let the RTS select among ready work units
    • Programmer should not specify what executes next, but can influence it via priorities
Realization of this model in Charm++

- Overdecomposed entities: chares
  - Chares are C++ objects
  - With methods designated as “entry” methods
    - Which can be invoked asynchronously by remote chares
  - Chares are organized into indexed collections
    - Each collection may have its own indexing scheme
      - 1D, ..7D,
      - Sparse
      - Bitvector or string as an index
  - Chares communicate via asynchronous method invocations
    - A[i].foo(....); A is the name of a collection, i is the index of the particular chare.
Overdecomposed Objects

Parallel Address Space
• Certain member functions of certain classes are globally visible
• Invocation of a member function may lead to communication
Message-driven Execution

Processor 0

Scheduler

Message Queue

A[..].foo(…)

Processor 1

Scheduler

Message Queue
Processor 0
Scheduler
Message Queue

Processor 1
Scheduler
Message Queue

Processor 2
Scheduler
Message Queue

Processor 3
Scheduler
Message Queue

8/6/15
ATRESC
Empowering the RTS

- The Adaptive RTS can:
  - Dynamically balance loads
  - Optimize communication:
    - Spread over time, async collectives
  - Automatic latency tolerance
  - Prefetch data with almost perfect predictability
Benefits in Charm++

- Over-decomposition
- Message-driven execution
- Migratability
- Introspective and adaptive runtime system

Scalable Tools
- Automatic overlap of Communication and Computation
- Perfect prefetch
- Compositionality
- Fault Tolerance
- Dynamic load balancing (topology-aware, scalable)
- Temperature/Power/Energy Optimizations

Emulation for Performance Prediction

Over-decomposition leads to message-driven execution, which in turn leads to Migratability and Introspective and adaptive runtime system.
Benefits in Charm++

- Over-decomposition
- message-driven execution
- Migratability
- Introspective and adaptive runtime system

- Scalable Tools
  - Automatic overlap of Communication and Computation
  - Perfect prefetch
  - compositionality
  - Emulation for Performance Prediction
  - Fault Tolerance
  - Dynamic load balancing (topology-aware, scalable)
  - Temperature/Power/Energy Optimizations

8/6/15  ATPESC
Utility for Multi-cores, Many-cores, Accelerators:

- Objects connote and promote locality
- Message-driven execution
  - A strong principle of prediction for data and code use
  - Much stronger than principle of locality
    - Can use to scale memory wall:
    - Prefetching of needed data:
      - into scratch pad memories, for example
Impact on communication

• Current use of communication network:
  – Compute-communicate cycles in typical MPI apps
  – So, the network is used for a fraction of time,
  – and is on the critical path

• So, current communication networks are over-engineered for by necessity
Impact on communication

• With overdecomposition
  – Communication is spread over an iteration
  – Also, adaptive overlap of communication and computation
Decomposition Challenges

• Current method is to decompose to processors
  – But this has many problems
  – Deciding which processor does what work in detail is difficult at large scale

• Decomposition should be independent of number of processors – enabled by object based decomposition

• Adaptive scheduling of the objects on available resources by the RTS
Decomposition Independent of numCores

• Rocket simulation example under traditional MPI

<table>
<thead>
<tr>
<th>Solid</th>
<th>Solid</th>
<th>...</th>
<th>Solid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fluid</td>
<td>Fluid</td>
<td></td>
<td>Fluid</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td>P</td>
</tr>
</tbody>
</table>

• With migratable-objects:

<table>
<thead>
<tr>
<th>Solid(_1)</th>
<th>Solid(_2)</th>
<th>Solid(_3)</th>
<th>...</th>
<th>Solid(_n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fluid(_1)</td>
<td>Fluid(_2)</td>
<td>Fluid(_3)</td>
<td>...</td>
<td>Fluid(_m)</td>
</tr>
</tbody>
</table>

– Benefit: load balance, communication optimizations, modularity
Compositionality

• It is important to support parallel composition
  – For multi-module, multi-physics, multi-paradigm applications...

• What I mean by parallel composition
  – B || C where B, C are independently developed modules
  – B is parallel module by itself, and so is C
  – Programmers who wrote B were unaware of C
  – No dependency between B and C

• This is not supported well by MPI
  – Developers support it by breaking abstraction boundaries
    • E.g., wildcard recvs in module A to process messages for module B
  – Nor by OpenMP implementations:
Without message-driven execution (and virtualization), you get either:

Space-division
OR: Sequentialization
Parallel Composition: A1; (B || C); A2

Recall: Different modules, written in different languages/paradigms, can overlap in time and on processors, without programmer having to worry about this explicitly.
So, What is Charm++?

• Charm++ is a way of parallel programming based on
  – Objects
  – Overdecomposition
  – Message
  – Asynchrony
  – Migratability
  – Runtime system
• **Charm++ Basics:**
• **Structured Dagger Notation**
• **Designing Charm++ programs, with application case studies**
Hello World Example

**hello.ci file**

```ci
mainmodule hello {
    mainchare Main {
        entry Main(CkArgMsg *m);
    }
};
```

**hello.cpp file**

```cpp
#include <stdio.h>
#include "hello.decl.h"

class Main : public CBase_Main {
    public: Main(CkArgMsg* m) {
        ckout << "Hello World!" << endl;
        CkExit();
    }
};

#include "hello.def.h"
```
Hello World with Chares

```cpp
#include <stdio.h>
#include "hello.decl.h"

class Main : public CBase_Main {
    public: Main(CkArgMsg* m) {
        CProxy_Singleton::ckNew();
    }
};

class Singleton : public CBase_Singleton {
    public: Singleton() {
        cout << "Hello World!" << endl;
        CkExit();
    }
};

#include "hello.def.h"
```

```ci
mainmodule hello {
    mainchare Main {
        entry Main(CkArgMsg *m);
    };
    chare Singleton {
        entry Singleton();
    };
};
```
Compiling a Charm++ Program
Building Charm++

- `git clone http://charm.cs.uiuc.edu/gerrit/charm`
- `./build <TARGET> <ARCH> <OPTS>`
- `TARGET = Charm++, AMPI, bgampi, LIBS etc.`
- `ARCH = net-linux-x86_64, multicore-darwin-x86_64, pamilrts-bluegeneq etc.`
- `OPTS = –with-production, –enable-tracing, xlc, smp, -j8 etc.`
Hello World Example

- **Compiling**
  - charmc hello.ci
  - charmc -c hello.C
  - charmc -o hello hello.o

- **Running**
  - ./charmrun +p7 ./hello
  - The +p7 tells the system to use seven cores
Charm++ File structure

- C++ objects (including Charm++ objects)
  - Defined in regular .h and .C files
- Chare objects, entry methods (asynchronous methods)
  - Defined in .ci file
  - Implemented in the .C file
Charm Interface: Modules

- Charm++ programs are organized as a collection of modules.
- Each module has one or more chares.
- The module that contains the *main chare*, is declared as the main module.
- Each module, when compiled, generates two files: `MyModule.decl.h` and `MyModule.def.h`.

```c
[main]module MyModule {
    //... chare definitions ...
};
```
Charm Interface: Chares

- Chares are parallel objects that are managed by the RTS
- Each chare has a set *entry methods*, which are asynchronous methods that may be invoked remotely
- The following code, when compiled, generates a C++ class `CBase_MyChare` that encapsulates the RTS object
- This generated class is extended and implemented in the `.C` file

**.ci file**

```c
[main]chare MyChare {
   //... entry method definitions ...
};
```

**.C file**

```c
class MyChare : public CBase_MyChare {
   //... entry method implementations ...
};
```
Entry methods are C++ methods that can be remotely and asynchronously invoked by another chare.

**.ci file:**

```cpp
entry MyChare(); /* constructor entry method */
entry void foo();
entry void bar(int param);
```

**.C file:**

```cpp
MyChare::MyChare() { /*... constructor code ...*/ }

MyChare::foo() { /*... code to execute ...*/ }

MyChare::bar(int param) { /*... code to execute ...*/ }
```
Execution begins with the mainchare’s constructor.

The mainchare’s constructor takes a pointer to a system-defined class `CkArgMsg`.

`CkArgMsg` contains `argv` and `argc`.

The mainchare will typically create some additional chares.
Creating a Chare

- A chare declared as `chare MyChare {...}`; can be instantiated by the following call:

  ```cpp
  CProxy<MyChare>::ckNew(... constructor arguments ...);
  ```

- To communicate with this class in the future, a `proxy` to it must be retained

  ```cpp
  CProxy<MyChare> proxy =
  CProxy<MyChare>::ckNew(... constructor arguments ...);
  ```
Chare Proxies

- A chare’s own proxy can be obtained through a special variable `thisProxy`.
- Chare proxies can also be passed so chares can learn about others.
- In this snippet, `MyChare` learns about a chare instance `main`, and then invokes a method on it:

```ci file
entry void foobar2(CProxy_Main main);
```

```C file
MyChare::foobar2(CProxy_Main main) {
    main.foo();
}
```
Charm Termination

- There is a special system call `CkExit()` that terminates the parallel execution on all processors (but it is called on one processor) and performs the requisite cleanup.
- The traditional `exit()` is insufficient because it only terminates one process, not the entire parallel job (and will cause a hang).
- `CkExit()` should be called when you can safely terminate the application (you may want to synchronize before calling this).
Chare Creation Example: .ci file

```plaintext
mainmodule MyModule {
    mainchare Main {
        entry Main(CkArgMsg *m);
    }
}

chare Simple {
    entry Simple(int x, double y);
}

```
#include <stdio.h>
#include "MyModule.decl.h"

class Main : public CBase_Main {
public: Main(CkArgMsg* m) {
    ckout << "Hello World!" << endl;
    if (m->argc > 1) ckout << "Hello " << m->argv[1] << "!!!" << endl;
    double pi = 3.1415;
    CProxy_Simple::ckNew(12, pi);
};
};
class Simple : public CBase_Simple {
public: Simple(int x, double y) {
    ckout << "Hello from a simple chare running on " << CkMyPe() << endl;
    ckout << "Area of a circle of radius" << x << " is " << y*x*x << endl;
    CkExit();
};

#include "MyModule.def.h"
Asynchronous Methods

- Entry methods are invoked by performing a C++ method call on a chare’s proxy

```cpp
CProxy_MyChare proxy =
    CProxy_MyChare::ckNew(... constructor arguments ...);

proxy.foo();
proxy.bar(5);
```

- The `foo` and `bar` methods will then be executed with the arguments, wherever the created chare, `MyChare`, happens to live
- The policy is one-at-a-time scheduling (that is, one entry method on one chare executes on a processor at a time)
Asynchronous Methods

- Method invocation is not ordered (between chares, entry methods on one chare, etc.)!
- For example, if a chare executes this code:

```cpp
CProxy_MyChare proxy = CProxy_MyChare::ckNew();
proxy.foo();
proxy.bar(5);
```

- These prints may occur in any order

```cpp
MyChare::foo() {
    cout << "foo executes" << endl;
}

MyChare::bar(int param) {
    cout << "bar executes with " << param << endl;
}
```
Asynchronous Methods

- For example, if a chare invokes the same entry method twice:
  ```cpp
  proxy.bar(7);
  proxy.bar(5);
  ```

- These may be delivered in **any** order
  ```cpp
  MyChare::bar(int param) {
    cout << "bar executes with " << param << endl;
  }
  ```

- **Output**
  ```
  bar executes with 5
  bar executes with 7
  ```
  **OR**
  ```
  bar executes with 7
  bar executes with 5
  ```
Asynchronous Example: .ci file

```
mainmodule MyModule {
    mainchare Main {
        entry Main(CkArgMsg *m);
    };
    chare Simple {
        entry Simple(double y);
        entry void findArea(int radius, bool done);
    };
};
```
Asynchronous Example: .C file

Does this program execute correctly?

```c
struct Main : public CBase_Main {
    Main(CkArgMsg* m) {
        double pi = 3.1415;
        CProxy_Simple sim = CProxy_Simple::ckNew(pi);
        for (int i = 1; i < 10; i++) sim.findArea(i, false);
        sim.findArea(10, true);
    }
};

struct Simple : public CBase_Simple {
    float y;
    Simple(double pi) {
        y = pi;
        cout << "Hello from a simple chare running on " << CkMyPe() << endl;
    }
    void findArea(int r, bool done) {
        cout << "Area of a circle of radius" << r << " is " << y*r*r << endl;
        if (done) CkExit();
    }
};
```
You can pass basic C++ types to entry methods (int, char, bool, etc.)

C++ STL data structures can be passed by including pup_stl.h

Arrays of basic data types can also be passed like this:

**.ci file:**

```ci
entry void foobar(int length, int data[length]);
```

**.C file:**

```c
MyChare::foobar(int length, int* data) {
   // ... foobar code ...
}
```
Collections of Objects: Concepts

- Objects can be grouped into indexed collections
- Basic examples
  - Matrix block
  - Chunk of unstructured mesh
  - Portion of distributed data structure
  - Volume of simulation space
- Advanced Examples
  - Abstract portions of computation
  - Interactions among basic objects or underlying entities
Collections of Objects

- Structured: 1D, 2D, ..., 6D
- Unstructured: Anything hashable
Collections of Objects

- Structured: 1D, 2D, . . . , 6D
- Unstructured: Anything hashable
- Dense
- Sparse
Collections of Objects

- Structured: 1D, 2D, ..., 6D
- Unstructured: Anything hashable
- Dense
- Sparse
- Static - all created at once
- Dynamic - elements come and go
mainmodule arr {

    mainchare Main {
        entry Main(CkArgMsg*);
    }

    array [1D] hello {
        entry hello(int);
        entry void printHello();
    }
}
#include "arr.decl.h"

struct Main : CBase_Main {
    Main(CkArgMsg* msg) {
        int arraySize = atoi(msg->argv[1]);
        CProxy_hello p = CProxy_hello::ckNew(arraySize, arraySize);
        p[0].printHello();
    }
};

struct hello : CBase_hello {
    hello(int n) : arraySize(n) {
    }
    hello(CkMigrateMessage*) {
    }
    void printHello() {
        CkPrintf("PE[%d]: hello from p[%d]\n", CkMyPe(), thisIndex);
        if (thisIndex == arraySize - 1) CkExit();
        else thisProxy[thisIndex + 1].printHello();
    }
    private:
    int arraySize;
};

#include "arr.def.h"
Hello World Array Projections Timeline View

- Add -tracemode projections to link line to enable tracing
- Run Projections tool to load trace log files and visualize performance

arrayHello on BG/Q 16 Nodes, mode c16, 1024 elements (4 per process)
Declaring a Chare Array

.ci file:

```c
array [1d] foo {
    entry foo(); // constructor
    // ... entry methods ...
}
array [2d] bar {
    entry bar(); // constructor
    // ... entry methods ...
}
```

.C file:

```c
struct foo : public CBase_foo {
    foo() {}
    foo(CkMigrateMessage*) {}
    // ... entry methods ...
};
struct bar : public CBase_bar {
    bar() {}
    bar(CkMigrateMessage*) {}
};
```
Constructing a Chare Array

- Constructed much like a regular chare
- The size of each dimension is passed to the constructor

```cpp
void someMethod() {
    CProxy_foo::ckNew(10);
    CProxy_bar::ckNew(5, 5);
}
```

- The proxy may be retained:

```cpp
CProxy_foo myFoo = CProxy_foo::ckNew(10);
```

- The proxy represents the entire array, and may be indexed to obtain a proxy to an individual element in the array

```cpp
myFoo[4].invokeEntry();
```
**thisIndex**

- 1d: `thisIndex` returns the index of the current chare array element
- 2d: `thisIndex.x` and `thisIndex.y` returns the indices of the current chare array element

`.ci` file:

```c
array [1d] foo {
    entry foo();
}
```

`.C` file:

```c
struct foo : public CBase_foo {
    foo() {
        CkPrintf("array index = %d", thisIndex);
    }
};
```
System knows how to ‘find’ objects efficiently:

\[(\text{collection, index}) \rightarrow \text{processor}\]

Applications can specify a mapping, or use simple runtime-provided options (e.g. blocked, round-robin)

Distribution can be static, or dynamic!

Key abstraction: application logic doesn’t change, even though performance might
Collections of Objects: Runtime Service

- Can develop and test logic in objects separately from their distribution
- Separation in time: make it work, then make it fast
- Division of labor: domain specialist writes object code, computationalist writes mapping
- Portability: different mappings for different systems, scales, or configurations
- Shared progress: improved mapping techniques can benefit existing code
Collections of Objects
Collective Communication Operations

- Point-to-point operations involve only two objects
- Collective operations that involve a collection of objects
- Broadcast: calls a method in each object of the array
- Reduction: collects a contribution from each object of the array
- A spanning tree is used to send/receive data

![Collective Communication Operations Diagram]
Broadcast

- A message to each object in a collection
- The chare array proxy object is used to perform a broadcast
- It looks like a function call to the proxy object
- From the main chare:

```cpp
CProxy_Hello helloArray = CProxy_Hello::ckNew(helloArraySize);
helloArray.foo();
```

- From a chare array element that is a member of the same array:

```cpp
thisProxy.foo();
```

- From any chare that has a proxy p to the chare array

```cpp
p.foo();
```
Reduction

- Combines a set of values: sum, max, aggregate
- Usually reduces the set of values to a single value
- Combination of values requires an operator
- The operator must be commutative and associative
- Each object calls contribute in a reduction
Reduction: Example

```c
mainmodule reduction {
  mainchare Main {
    entry Main(CkArgMsg* msg);
    entry [reductiontarget] void done(int value);
  };
  array [1D] Elem {
    entry Elem(CProxy_Main mProxy);
  };
}
```
```cpp
#include "reduction.decl.h"

const int numElements = 49;

class Main : public CBase_Main {
public:
    Main(CkArgMsg* msg) { CProxy_Elem::ckNew(thisProxy, numElements); }
    void done(int value) {
        CkAssert(value == numElements * (numElements - 1) / 2);
        CkPrintf("value: %d\n", value);
        CkExit();
    }
};

class Elem : public CBase_Elem {
public:
    Elem(CProxy_Main mProxy) {
        int val = thisIndex;
        CkCallback cb(CkReductionTarget(Main, done), mProxy);
        contribute(sizeof(int), &val, CkReduction::sum_int, cb);
    }
    Elem(CkMigrateMessage*) { }
};

#include "reduction.def.h"
```

Output:

value: 1176
Program finished.
Chares are reactive

• The way we described Charm++ so far, a chare is a reactive entity:
  ➢ If it gets this method invocation, it does this action,
  ➢ If it gets that method invocation then it does that action
  ➢ But what does it do?
  ➢ In typical programs, chares have a life-cycle
• How to express the life-cycle of a chare in code?
  ➢ Only when it exists
    * i.e. some chars may be truly reactive, and the programmer does not know the life cycle
  ➢ But when it exists, its form is:
    * Computations depend on remote method invocations, and completion of other local computations
    * A DAG (Directed Acyclic Graph)!
Structured Dagger (sdag)
The *when* construct

- sdag code is written in the `.ci` file
- It is like a script, with a simple language
- Important: The *when* construct
  - Declare the actions to perform when a method invocation is received
  - In sequence, it acts like a blocking receive

```java
entry void someMethod() {
    when entryMethod1(parameters) { block1 }
    when entryMethod2(parameters) { block2 }
    block3
}
```
The *serial* construct

- A sequential block of C++ code in the `.ci` file
- The keyword *serial* means that the code block will be executed without interruption/preemption
- Syntax: `serial <optionalString> { /*C++ code*/ }`
- The `<optionalString>` is just a tag for performance analysis
- Serial blocks can access all members of the class they belong to

```cpp
entry void method1(parameters) {
    when E(a)
    serial
       { thisProxy.invokeMethod(10, a);
          callSomeFunction(); } 
... 
}

entry void method2(parameters) {
    ... 
    serial "setValue" {
        value = 10;
    }
}
```
entry void someMethod() {
    serial { /* block1 */ }
    when entryMethod1(parameters) serial { /* block2 */ }
    when entryMethod2(parameters) serial { /* block3 */ }
};

- Sequentially execute:
  1. /* block1 */
  2. Wait for `entryMethod1` to arrive, if it has not, return control back to the Charm++ scheduler, otherwise, execute /* block2 */
  3. Wait for `entryMethod2` to arrive, if it has not, return control back to the Charm++ scheduler, otherwise, execute /* block3 */
Structured Dagger

The \textit{when} construct

- You can combine waiting for multiple method invocations
- Execute \textit{“code-block”} when $M1$ and $M2$ arrive
- You have access to \texttt{param1}, \texttt{param2}, \texttt{param3} in the code-block

\begin{verbatim}
When M1(int param1, int param2), M2(bool param3)
    { code block }
\end{verbatim}
• Structured Dagger can be used in any entry method (except for a constructor)
• For any class that has Structured Dagger in it you must insert:
  • The Structured Dagger macro:  
    
    `[ClassName]_SDAG_CODE`
  

Structured Dagger

Boilerplate

The .ci file:

```cpp
[mainchare,chare,array,...] MyFoo {
...
  entry void method(parameters) {
    // ... structured dagger code here ...
  };
  ...
}
```

The .cpp file:

```cpp
class MyFoo : public CBase {
  MyFoo_SDAG_Code/* insert SDAG macro */
public:
  MyFoo() {}
};
```
Structured Dagger
The \texttt{when} construct: refnum

- The \texttt{when} clause can wait on a certain reference number.
- If a reference number is specified for a \texttt{when}, the first parameter for the \texttt{when} must be the reference number.
- Semantics: the \texttt{when} will “block” until a message arrives with that reference number.

\begin{verbatim}
\texttt{when method1[100](int ref, bool param1)}
  \texttt{/* sdag block */}
\end{verbatim}

\begin{verbatim}
\texttt{serial {
  proxy.method1(200, false); \texttt{/* will not be delivered to the when */}
  proxy.method1(100, true); \texttt{/* will be delivered to the when */}
}}
\end{verbatim}
The *if-then-else* construct:

- Same as the typical C if-then-else semantics and syntax

```cpp
if (thisIndex.x == 10) {
    when method1[block](int ref, bool someVal) /* code block1 */
} else {
    when method2(int payload) serial {
        //... some C++ code
    }
}
```
Structured Dagger
The *for* construct

- The *for* construct:
  - Defines a sequenced *for* loop (like a sequential C for loop)
  - Once the body for the *i*th iteration completes, the *i* + 1 iteration is started

```cpp
for (iter = 0; iter < maxIter; ++iter) {
    when recvLeft[iter](int num, int len, double data[len])
        serial { computeKernel(LEFT, data); }
    when recvRight[iter](int num, int len, double data[len])
        serial { computeKernel(RIGHT, data); }
}
```

- *iter* must be defined in the class as a member

```cpp
class Foo : public CBase Foo {
    public: int iter;
};
```
• The `while` construct:
  - Defines a sequenced `while` loop (like a sequential C while loop)

```c
while (i < numNeighbors) {
    when recvData(int len, double data[len]) {
        serial {
            /* do something */
        }
        when method1() /* block1 */
        when method2() /* block2 */
    }
    serial { i++; }
}
```
Structured Dagger

The overlap construct

- The overlap construct:
  - By default, Structured Dagger constructs are executed in a sequence
  - overlap allows multiple independent constructs to execute in any order
  - Any constructs in the body of an overlap can happen in any order
  - An overlap finishes when all the statements in it are executed
  - Syntax: overlap { /* sdag constructs */ }

What are the possible execution sequences?

```plaintext
serial { /* block1 */ }
overlap {
    serial { /* block2 */ }
    when entryMethod1[100](int ref num, bool param1) /* block3 */
    when entryMethod2(char myChar) /* block4 */
}
serial { /* block5 */ }
```
• *Overlap* can be used to regain some asynchrony within a chare
• But it is constrained
• More disciplined programming,
• with fewer race conditions
Structured Dagger

The \textit{forall} construct

- The \textit{forall} construct:
  - Has “do-all” semantics: iterations may execute in any order
  - Syntax:
    \begin{verbatim}
    forall [<ident>] (<min> : <max>, <stride>) <body>
    \end{verbatim}
  - The range from <min> to <max> is inclusive

\begin{verbatim}
forall [block] (0 : numBlocks − 1, 1) {
  when method1[block](int ref, bool someVal) /* code block1 */
}
\end{verbatim}

- Assume \textit{block} is declared in the class as \textit{public: int block;}

1-D decomposition: each chare object owns a strip
Need to exchange top and bottom boundaries
mainmodule jacobi1d {
  readonly CProxy Main mainProxy;
  readonly int blockDimX;
  readonly int numChares;

  mainchare Main {
    entry Main(CkArgMsg ∗m);
  };

  array [1D] Jacobi {
    entry Jacobi(void);
    entry void recvGhosts(int iter, int dir, int size, double gh[size]);
    entry [reductiontarget] void isConverged(bool result);
    entry void run() {
      // ... main loop (next slide) ...
    }
  }
};
while (!converged) {
    serial "send_to_neighbors" {
        iter++; top = (thisIndex+1)\%numChares; bottom = ...;
        thisProxy(top).recvGhosts(iter, BOTTOM, arrayDimY, &value[1][1]);
        thisProxy(bottom).recvGhosts(iter, TOP, arrayDimY, &value[blockDimX][1]);
    }
    for(imsg = 0; imsg < neighbors; imsg++)
        when recvGhosts[iter] (int iter, int dir, int size, double gh[size])
            serial "update_boundary" {
                int row = (dir == TOP) ? 0 : blockDimX+1;
                for(int j=0; j<size; j++) value[row][j+1] = gh[j];
            }
    serial "do_work" {
        conv = check_and_compute(); // conv: a boolean indicating local convergence
        CkCallback cb = CkCallback(CkReductionTarget(Jacobi, isConverged), thisProxy);
        Contribute(sizeof(bool), &conv, CkReduction::logical_and, cb);
    }
    when isConverged(bool result) serial "check_converge" {
        converged = result; if (result && thisIndex == 0) CkExit();
    }
}
while (!converged) {
    serial "send_to_neighbors" {
        iter++;  
top = (thisIndex+1)%numChares;  bottom = ...;
        thisProxy(top).recvGhosts(iter, BOTTOM, arrayDimY, &value[1][1]);
        thisProxy(bottom).recvGhosts(iter, TOP, arrayDimY, &value[blockDimX][1]);
    }
    for(imsg = 0; imsg < neighbors; imsg++)
        when recvGhosts[iter] (int iter, int dir, int size, double gh[size])
            serial "update_boundary" {
                int row = (dir == TOP) ? 0 : blockDimX+1;
                for(int j=0; j<size; j++) value[row][j+1] = gh[j];
            }
    serial "do_work" {
        conv = check_and_compute();   // conv: a boolean indicating local convergence
        CkCallback cb = CkCallback(CkReductionTarget(Jacobi, isConverged), thisProxy);
        Contribute(sizeof(bool), &conv, CkReduction::logical_and, cb);
    }
    when isConverged(bool result) serial "check_converge" {
        converged = result; if (result && thisIndex == 0) CkExit();
    }
    if (iter % LBPERIOD == 0) {serial "start_lb" { AtSync();} when ResumeFromSync() {}
Grainsize

- Charm++ philosophy:
  - let the programmer decompose their work and data into coarse-grained entities
- It is important to understand what I mean by coarse-grained entities
  - You don’t write sequential programs that some system will auto-decompose
  - You don’t write programs when there is one object for each `float`
  - You consciously choose a grainsize, **BUT** choose it independent of the number of processors
    - Or parameterize it, so you can tune later
Crack Propagation

This is 2D, circa 2002... but shows over-decomposition for unstructured meshes.

Decomposition into 16 chunks (left) and 128 chunks, 8 for each PE (right). The middle area contains cohesive elements. Both decompositions obtained using Metis. Pictures: S. Breitenfeld, and P. Geubelle
Grainsize example: NAMD

- High Performing examples: (objects are the work-data units in Charm++)
- On Blue Waters, 100M atom simulation,
  - 128K cores (4K nodes), 5,510,202 objects
- Edison, Apoa1(92K atoms)
  - 4K cores, 33124 objects
- Hopper, STMV, 1M atoms,
  - 15,360 cores, 430,612 objects
Grainsize: Weather Forecasting in BRAMS

- **Brams**: Brazilian weather code (based on RAMS)
- **AMPI version** (Eduardo Rodrigues, with Mendes, J. Panetta, ..)

Instead of using 64 work units on 64 cores, used 1024 on 64
Working definition of grainsize: amount of computation per remote interaction

Choose grainsize to be just large enough to amortize the overhead
Grainsize in a common setting

Jacobi3D running on JYC using 64 cores on 2 nodes

2048x2048x2048 (total problem size)

2 MB/chare, 256 objects per core
Rules of thumb for grainsize

• Make it as small as possible, as long as it amortizes the overhead

• More specifically, ensure:
  – *Average* grainsize is greater than $k \cdot v$ (say 10v)
  – No single grain should be allowed to be too large
    • Must be smaller than $T/p$, but actually we can express it as
      – Must be smaller than $k \cdot m \cdot v$ (say 100v)

• Important corollary:
  – You can be at close to optimal grainsize without having to think about $P$, the number of processors
Charm++ Applications as case studies

Only brief overview today
NAMD: Biomolecular Simulations

- Collaboration with K. Schulten
- With over 50,000 registered users
- Scaled to most top US supercomputers
- In production use on supercomputers and clusters and desktops
- Gordon Bell award in 2002

Recent success: Determination of the structure of HIV capsid by researchers including Prof Schulten
Molecular Dynamics: NAMD

- Collection of [charged] atoms
  - With bonds
  - Newtonian mechanics
  - Thousands to millions atoms

- At each time-step
  - Calculate forces on each atom
    - Bonds
    - Non-bonded: electrostatic and van der Waal’s
      - Short-distance: every timestep
      - Long-distance: using PME (3D FFT)
      - Multiple Time Stepping: PME every 4 timesteps
  - Calculate velocities
  - Advance positions

Challenge: femtosecond time-step, millions needed!
Hybrid Decomposition

Object Based Parallelization for MD: Force Decomp. + Spatial Decomp.

- We have many objects to load balance:
  - Each diamond can be assigned to any proc.
  - Number of diamonds (3D):
    - $14 \cdot \text{Number of Cells}$
Parallelization using Charm++
Sturdy design!

• This design,
  – done in 1995 or so, running on 12 node HP cluster

• Has survived
  – With minor refinements

• Until today
  – Scaling to 500,000+ cores on Blue Waters!
  – 300,000 Cores of Jaguar, or BlueGene/P

We are developing the NAMD project, for instance, on a cluster of twelve HP 735/125 workstations connected with an ATM, or asynchro-
Shallow valleys, high peaks, nicely overlapped PME

Apo-A1, on BlueGene/L, 1024 procs

Time intervals on X axis, activity added across processors on Y axis
NAMD strong scaling on Titan Cray XK7, Blue Waters Cray XE6, and Mira IBM Blue Gene/Q for 21M and 224M atom benchmarks.
ChaNGa: Parallel Gravity

• Collaborative project (NSF)
  – with Tom Quinn, Univ. of Washington

• Gravity, gas dynamics

• Barnes–Hut tree codes
  – Oct tree is natural decomp
  – Geometry has better aspect ratios, so you “open” up fewer nodes
  – But is not used because it leads to bad load balance
  – Assumption: one-to-one map between sub-trees and PEs
  – Binary trees are considered better load balanced

Evolution of Universe and Galaxy Formation

With Charm++: Use Oct-Tree, and let Charm++ map subtrees to processors
ChaNGa: Cosmology Simulation

- Tree: Represents particle distribution
- TreePiece: object/char/containing particles

Collaboration with Tom Quinn UW
ChaNGa: Optimized Performance

- Asynchronous, highly overlapped, phases
- Requests for remote data overlapped with local computations
ChaNGa: a recent result
Episimdemics

• Simulation of spread of contagion
  – Code by Madhav Marathe, Keith Bisset, .. Vtech
  – Original was in MPI
• Converted to Charm++
  – Benefits: asynchronous reductions improved performance considerably
Simulating contagion over dynamic networks

EpiSimdemics

- Agent-based
- Realistic population data
- Intervention
- Co-evolving network, behavior and policy

\[ P = 1 - \exp(t \cdot \log(1 - I \cdot S)) \]
- \( t \): duration of co-presence
- \( I \): infectivity
- \( S \): susceptibility

1C. Barrett et al., “EpiSimdemics: An Efficient Algorithm for Simulating the Spread of Infectious Disease over Large Realistic Social Networks,” SC08
Strong scaling performance with the largest data set

Contiguous US population data
- XE6: the largest scale (352K cores)
- BG/Q: good scaling up to 128K cores
- Strong scaling helps timely reaction to pandemic
Using Charm++ virtualization, we can efficiently scale small (32 molecule) systems to thousands of processors.
Topology Aware Mapping of Objects

3D Torus of the machine

RealSpace

States

Planes

RealSpace Prisms perpendicular to Gspace Prisms

Density

Planes

States

PairCalculator

Rectangular Gspace Prisms

GSpace

$n_s$

$n_s$

$n_s$

$n_g$

$n_g$

$n_g$

block_size
Improvements by topological aware mapping of computation to processors

The simulation of the left panel, maps computational work to processors taking the network connectivity into account while the right panel simulation does not. The “black” or idle time processors spent waiting for computational work to arrive on processors is significantly reduced at left. (256waters, 70R, on BG/L 4096 cores)

Punchline: Overdecomposition into Migratable Objects created the degree of freedom needed for flexible mapping
OpenAtom Performance Sampler

Ongoing work on:
K-points

OpenAtom running WATER 256M 70Ry on various platforms

- Blue Gene/L
- Blue Gene/P
- Cray XT3

No. of cores vs. Timestep (secs/step) graph.
## MiniApps

<table>
<thead>
<tr>
<th>Mini-App</th>
<th>Features</th>
<th>Machine</th>
<th>Max cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMR</td>
<td>Overdecomposition, Custom array index, Message priorities, Load Balancing, Checkpoint restart</td>
<td>BG/Q</td>
<td>131,072</td>
</tr>
<tr>
<td>LeanMD</td>
<td>Overdecomposition, Load Balancing, Checkpoint restart, Power awareness</td>
<td>BG/P, BG/Q</td>
<td>131,072, 32,768</td>
</tr>
<tr>
<td>Barnes–Hut (n–body)</td>
<td>Overdecomposition, Message priorities, Load Balancing</td>
<td>Blue Waters</td>
<td>16,384</td>
</tr>
<tr>
<td>LULESH 2.02</td>
<td>AMPI, Overdecomposition, Load Balancing</td>
<td>Hopper</td>
<td>8,000</td>
</tr>
<tr>
<td>PDES</td>
<td>Overdecomposition, Message priorities, TRAM</td>
<td>Stampede</td>
<td>4,096</td>
</tr>
</tbody>
</table>
More MiniApps

<table>
<thead>
<tr>
<th>Mini-App</th>
<th>Features</th>
<th>Machine</th>
<th>Max cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D FFT</td>
<td>Interoperable with MPI</td>
<td>BG/P BG/Q</td>
<td>65,536 16,384</td>
</tr>
<tr>
<td>Random Access</td>
<td>TRAM</td>
<td>BG/P BG/Q</td>
<td>131,072 16,384</td>
</tr>
<tr>
<td>Dense LU</td>
<td>SDAG</td>
<td>XT5</td>
<td>8,192</td>
</tr>
<tr>
<td>Sparse Triangular Solver</td>
<td>SDAG</td>
<td>BG/P</td>
<td>512</td>
</tr>
<tr>
<td>GTC</td>
<td>SDAG</td>
<td>BG/Q</td>
<td>1,024</td>
</tr>
<tr>
<td>SPH</td>
<td></td>
<td>Blue Waters</td>
<td>–</td>
</tr>
</tbody>
</table>
A recently published book surveys seven major applications developed using Charm++

More info on Charm++: http://charm.cs.illinois.edu
Including the miniApps
Where are Exascale Issues?

• I didn’t bring up exascale at all so far..
  – Overdecomposition, migratability, asynchrony were needed on yesterday’s machines too
  – And the app community has been using them
  – But:
    • On *some* of the applications, and maybe without a common general-purpose RTS

• The same concepts help at exascale
  – Not just help, they are necessary, and adequate
  – As long as the RTS capabilities are improved

• We have to apply overdecomposition to all (most) apps
Relevance to Exascale

Intelligent, introspective, Adaptive Runtime Systems, developed for handling application’s dynamic variability, already have features that can deal with challenges posed by exascale hardware.
Fault Tolerance in Charm++/AMPI

- Four approaches available:
  - Disk-based checkpoint/restart
  - In-memory double checkpoint w auto. restart
  - Proactive object migration
  - Message-logging: scalable fault tolerance

- Common Features:
  - Easy checkpoint: migrate-to-disk
  - Based on dynamic runtime capabilities
  - Use of object-migration
  - Can be used in concert with load-balancing schemes

Demo at Tech Marketplace
Saving Cooling Energy

- Easy: increase A/C setting
  - But: some cores may get too hot
- So, reduce frequency if temperature is high (DVFS)
  - Independently for each chip
- But, this creates a load imbalance!
- No problem, we can handle that:
  - Migrate objects away from the slowed-down processors
  - Balance load using an existing strategy
  - Strategies take speed of processors into account
- Implemented in experimental version
  - SC 2011 paper, IEEE TC paper
- Several new power/energy-related strategies
  - PASA ‘12: Exploiting differential sensitivities of code segments to frequency change

Demo at Tech Marketplace
PARM: Power Aware Resource Manager

- Charm++ RTS facilitates malleable jobs
- PARM can improve throughput under a fixed power budget using:
  - overprovisioning (adding more nodes than conventional data center)
  - RAPL (capping power consumption of nodes)
  - Job malleability and moldability

![Diagram of PARM architecture](chart.png)
Summary

• Charm++ embodies an adaptive, introspective runtime system

• Many applications have been developed using it
  - NAMD, ChaNGa, Episimdemics, OpenAtom, ...
  - Many miniApps, and third-party apps

• Adaptivity developed for apps is useful for addressing exascale challenges
  - Resilience, power/temperature optimizations, ..

More info on Charm++:
http://charm.cs.illinois.edu
Including the miniApps

Overdecomposition  Asynchrony  Migratability