Migratable Objects and Task-Based Parallel Programming with Charm++

© 2018 L. V. Kale at the University of Illinois Urbana
Challenges in Parallel Programming

• Applications are getting more sophisticated
  • Adaptive refinement
  • 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
  • Strong scaling: run an application with same input data on more processors, and get better speedups
  • Weak scaling: larger datasets on more processors in the same time

• Hardware variability
  • Static/dynamic
  • Heterogeneity: processor types, process variation, etc.
  • Power/Temperature/Energy
  • Component failure
Our View

• To deal with these challenges, we must seek:
  • Not full automation
  • Not full burden on app-developers
  • But: a good division of labor between the system and app developers
    • Programmer: what to do in parallel, System: where, when

• Develop language driven by needs of real applications
  • Avoid “platonic” pursuit of “beautiful” ideas
  • Co-developed with NAMD, ChaNGa, OpenAtom, ...

• Pragmatic focus
  • Ground-up development, portability
  • Accessibility for a broad user base
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:
  • Over-decomposition, Migratability, Asynchrony
Over-Decomposition

• 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 application 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

• With over-decomposition and migratability:
  • 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++

• Over-decomposed 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, ..., 6D
      • 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
Global Object Space

Processor 0
- Scheduler
- Message Queue

Processor 1
- Scheduler
- Message Queue

Processor 2
- Scheduler
- Message Queue

Processor 3
- Scheduler
- Message Queue

L.V. Kale
Message-Driven Execution

A[23].foo(…)

Processor 0
Scheduler
Message Queue

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
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
Enabling CS technology of parallel objects and intelligent runtime systems has led to several CSE collaborative applications.
Summary: What Is Charm++?

• Charm++ is a way of parallel programming
• It is based on:
  • Objects
  • Over-decomposition
  • Asynchrony
    • Asynchronous method invocations
  • Migratability
  • Adaptive runtime system
• It has been co-developed synergistically with multiple CSE applications
Parallel Programming with Charm++

Grainsize

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

© 2018 L. V. Kale at the University of Illinois Urbana
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 independently 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
Working definition of grainsize:
amount of computation per remote interaction

Choose grainsize to be just large
eough 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

number of points per chare
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
Baseline: 64 Objects

Profile of Usage for Processors 0-63
Time per Step: 46s
Over-Decomposition: 1024 Objects

Profile of Usage for Processors 0-63
Time per Step: 33s

Benefits from communication/computation overlap
With Load Balancing: 1024 objects

Usage Profile for Processors 0-63
Time per Step: 27s

No over-decomp (64 threads) 46 sec
+ Over-decomposition (1024 threads) 33 sec
+ Load balancing (1024 threads) 27 sec
Parallel Programming with Charm++

Benefits
Charm++ Benefits

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

L.V. Kale
Locality and Prefetch

• 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 scratchpad memories, for example
Impact on Communication

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

• Current communication networks are over-engineered by necessity
Impact on Communication

• With over-decomposition:
  • Communication is spread over an iteration
  • Adaptive overlap of communication and computation
Communication Data from Chombo

Work by Phil Miller

Bytes Sent Over Time

Chombo with reductions

Chombo on Charm (experimental)
Decomposition Challenges

• Current method is to decompose to processors
  • 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

• Let runtime system (RTS) assign objects to available resources adaptively
Decomposition Independent of numCores

• Rocket simulation example under traditional MPI

• With migratable objects:

  Solid \_1 \\
  Fluid \_1

  Solid \_2 \\
  Fluid \_2

  \ldots

  Solid \_P \\
  Fluid \_P

  Solid \_1 \ldots \_n

  Fluid \_1 \ldots \_m

• 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
Recall: different modules, written in different languages/paradigms, can overlap in time and on processors, without programmer having to worry about this explicitly.

Parallel composition: \( A_1; (B \parallel C); A_2 \)
Chares: Message-Driven Objects
Defining Chares and Understanding Asynchrony

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

© 2018 L. V. Kale at the University of Illinois Urbana
Hello World with Chares

**hello.ci**

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

**ci file is processed to generate code for classes such as Cbase_Main, Cbase_Singleton, Cproxy_Singleton**

**hello.cpp**

```cpp
#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"
```

L.V.Kale
Charm++ File Structure

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

**Hello World Example**

- **Compiling**
  - charmc hello.ci
  - charmc -c hello.cpp
  - charmc -o hello hello.o
- **Running**
  - ./charmrun +p7 ./hello
  - The +p7 tells the system to use seven cores
Compiling a Charm++ Program

1. **.ci** interface file
2. **charmcc** (charmui)
3. **.decl.h** temp. file
4. **.h** header file
5. **C or .cpp** source file
6. **.O** object file
7. **#include “xxx.decl.h”**
8. **#include “xxx.h”**
9. **#include “xxx.def.h”**
Hello World with Chares

**hello.ci**

```c
mainmodule hello {
    mainchare Main {
        entry Main(CkArgMsg *m);
    };
    chare Singleton {
        entry Singleton();
    };
};
```

**hello.cpp**

```c
#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"
```
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)
Entry Method Invocation Example: .ci file

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

    chare Simple {
        entry Simple(double y);
        entry void findArea(int radius, bool done);
    };
};
```
Does This Program Execute Correctly?

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

struct Simple : public CBase_Simple {
    double y;
    Simple(double pi) { y = pi; }
    void findArea(int r, bool done) {
        cout << "Area:" << y*r*r << endl;
        if (done) CkExit();
    }
};
```
No! Methods Are Asynchronous

• If a chare sends multiple entry method invocations

```cpp
sim.findArea(1, false);
...
sim.findArea(10, true);
```

• These may be delivered in any order

```cpp
Simple::findArea(int r, bool done){
    cout << "Area:" << y*r*r << endl;
    if (++count == 10) CkExit();
};
```

• Output:

<table>
<thead>
<tr>
<th>Area: 254.34</th>
<th>Area: 28.26</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area: 200.96</td>
<td>Area: 78.50</td>
</tr>
<tr>
<td>Area: 28.26</td>
<td>Area: 3.14</td>
</tr>
<tr>
<td>Area: 200.96</td>
<td>Area: 113.04</td>
</tr>
<tr>
<td>Area: 28.26</td>
<td>Area: 314.00</td>
</tr>
<tr>
<td>Area: 153.86</td>
<td>Area: 314.00</td>
</tr>
<tr>
<td>Area: 50.24</td>
<td>or</td>
</tr>
<tr>
<td>Area: 78.50</td>
<td></td>
</tr>
<tr>
<td>Area: 3.14</td>
<td></td>
</tr>
<tr>
<td>Area: 12.56</td>
<td></td>
</tr>
<tr>
<td>Area: 153.86</td>
<td></td>
</tr>
<tr>
<td>Area: 314.00</td>
<td></td>
</tr>
</tbody>
</table>
Chare Arrays: Collection of Chares
Chare Arrays

• Indexed collections of chares
  • Every item in the collection has a unique index and proxy
  • Can be indexed like an array or by an arbitrary object
  • Can be sparse or dense
  • Elements may be dynamically inserted and deleted
  • Elements are distributed across the available processors
    • May be migrated to other nodes by the user or the runtime

• For many scientific applications, collections of chares are a convenient abstraction
Declaring a Chare Array

.ci file:

```plaintext
chare foo {
    entry foo(); // constructor
    // ... entry methods ...
}
chare bar {
    entry bar(); // constructor
    // ... entry methods ...
}
```
Constructing a Chare Array

• Constructed much like a regular chare, using ckNew
• The size of each dimension is passed to the constructor at the end

```c
void someMethod() {
    CProxy_foo myFoo = CProxy_foo::ckNew(<params>, 10); // 1d, size 10
    CProxy_bar myBar = CProxy_bar::ckNew(<params>, 5, 5); // 2d, size 5x5
}
```

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

```c
myFoo[4].invokeEntry(...);
myBar(2,4).method3(...);
```
thisIndex

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

**.ci file:**

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

**.cpp file:**

```c
struct foo : public CBase_foo {
    foo() {
        cout << "array index: " << thisIndex;
    }
};
```
Chare Array: Hello Example

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

#include "arr.def.h"
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 a chare array element that is a member of the same array:

```java
thisProxy.foo();
```

• From any chare that has a proxy p to the chare array:

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

• Combines a set of values:
  • The operator must be commutative and associative
    • sum, max, ...

• Each object calls `contribute` in a reduction
Reduction: Example

```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) { 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);
    }
};
#include "reduction.def.h"
```

Output
value: 1176
Program finished.
mainmodule reduction {
    mainchare Main {
        entry Main(CkArgMsg* msg);
        entry [reductiontarget] void done(int value);
    }
    array [1D] Elem {
        entry Elem(CProxy_Main mProxy);
    }
}
Reduction Example : callback to a chare-array element

```cpp
#include "reduction.decl.h"
const int numElements = 49;
class Main : public CBase_Main {
public:
    Main(CkArgMsg* msg) { CProxy_Elem::ckNew(thisProxy, numElements); }
};

class Elem : public CBase_Elem {
public:
    Elem(CProxy_Main mProxy) {
        int val = thisIndex;
        CkCallback cb(CkReductionTarget(Elem, done), thisProxy[0]);
        contribute(sizeof(int), &val, CkReduction::sum_int, cb);
    }
    void done(int value) { CkPrintf("value: %d\n", value); CkExit(); }
};
#include "reduction.def.h"
```
Reduction Example: callback to a whole chare-array

```cpp
#include "reduction.decl.h"
const int numElements = 49;
class Main : public CBase_Main {
public:
    Main(CkArgMsg* msg) { CProxy_Elem::ckNew(thisProxy, numElements); }
};
class Elem : public CBase_Elem {
public:
    Elem(CProxy_Main mProxy) {
        int val = thisIndex;
        CkCallback cb(CkReductionTarget(Elem, tellAll), thisProxy);
        contribute(sizeof(int), &val, CkReduction::sum_int, cb);
    }
    void tellAll(int value) {
        CkPrintf("Elem[%d]: value: %d\n", thisIndex, value);
    }
};
#include "reduction.def.h"
```

Broadcast callback
Chare Arrays view
Dynamic Load Balancing in Charm++

Invoking Load Balancers, and serialization via PUPs

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Dynamic Load Balancing

• Object-based decomposition (i.e., virtualized decomposition) helps
  • Charm++ RTS reassigns objects to PEs to balance load
  • But how does the RTS decide?
    • Multiple strategy options
    • E.g., just move objects away from overloaded processors to underloaded processors
• How is load determined?
Measurement Based Load Balancing

• Principle of persistence
  • Object communication patterns and computational loads tend to persist over time
  • In spite of dynamic behavior
    • Abrupt but infrequent changes
    • Slow and small changes
  • Recent past is a good predictor of near future

• Runtime instrumentation
  • Measures communication volume and computation time

• Measurement-based load balancers
  • Measure load information for chares
  • Periodically use the instrumented database to make new decisions and migrate objects
  • Many alternative strategies can use the database
Using the Load Balancer

• Link a LB module
  • -module <strategy>
  • RefineLB, NeighborLB, GreedyCommLB, others
  • EveryLB will include all load balancing strategies

• Compile time option (specify default balancer)
  • -balancer RefineLB

• Runtime option (override default)
  • +balancer RefineLB
Instrumentation

• By default, instrumentation is enabled
  • Automatically collects load information

• Sometimes, you want LB decisions to be based only on a portion of your program
  • To disable by default, provide runtime argument +LBOff
  • To toggle instrumentation in code, use LBTurnInstrumentOn() and LBTurnInstrumentOff()
Code to Use Load Balancing

• Set usesAtSync = true; in chare constructor

• Insert AtSync() call at a natural barrier
  • Call from every chare in all collections
  • Does not block

• Implement ResumeFromSync() to resume execution
  • A typical ResumeFromSync() contributes to a reduction

L.V.Kale
// Synchronize at every iteration: Main starts next iteration
void Main::endIter(err) { if (err < T) CkExit();
    else stencilProxy.sendBoundaries(); }

// Assume a 1D Stencil chare array with near neighbor communication
void Stencil::sendBoundaries() {
    thisProxy(wrap(x-1)).updateGhost(RIGHT, left_ghost);
    thisProxy(wrap(x+1)).updateGhost(LEFT, right_ghost);
}

void Stencil::updateGhost(int dir, double ghost) { 
    updateBoundary(dir, ghost);
    if (++remoteCount == 2) {
        remoteCount = 0;
        doWork(); } }
Example: Stencil (cont.)

```c++
void Stencil::doWork() {
    e = (computeKernel() < DELTA);

    contribute(8, e, CkCallback(CkReductionTarget(Main, endIter), mainProxy));
}
```
Serialization and PUP

• How can the RTS move arbitrary objects across nodes?
• Charm++ has a framework for serializing data called PUP
• **PUP**: Pack and Unpack
• With PUP, chares become serializable and can be transported to memory, disk, or another processor
class MyChare :
public Cbase_MyChare {
    int a;
    float b;
    char c;
    double localArray[LOCAL_SIZE];
};

void pup(PUP::er &p) {
    p | a;
    p | b;
    p | c;
    p(localArray, LOCAL_SIZE);
}
Writing an Advanced PUP Routine

class MyChare : public Cbase_MyChare {
    int heapArraySize;
    float* heapArray;
    MyClass* pointer;
};

void pup(PUP::er &p) {
    p | heapArraySize;
    if (p.isUnpacking()) {
        heapArray = new float[heapArraySize];
    }
    p(heapArray, heapArraySize);
    bool isNull = !pointer;
    p | isNull;
    if (!isNull) {
        if (p.isUnpacking()) {
            pointer = new MyClass();
        }
        p | *pointer;
    }
PUP Uses

• Moving objects for load balancing
• Marshalling user defined data types
  • When using a type you define as a parameter for an entry method
  • Type has to be serialized to go over network, uses PUP for this
  • Can add PUP to any class, doesn’t have to be a chare
• Serializing for storage
Split Execution: Checkpoint Restart

• Can use to stop execution and resume later
  • The job runs for 5 hours, then will continue in new allocation another day!
• We can use PUP for this!
• Instead of migrating to another PE, just “migrate” to disk
• Bonus: in a well-written program, you can resume on a different number of nodes
  • Example: Original run on 100 nodes, checkpointed, can be resumed on 80 nodes (say because that’s what was available).
How to Enable Split Execution

• Call to checkpoint the application is made in the main chare at a synchronization point

• log_path is file system path for checkpoint

• Callback cb called when checkpoint (or restart) is done
  • For restart, user needs to provide argument +restart and path of checkpoint file at runtime

CkCallback cb (CkIndex_Hello:SayHi(), helloProxy);
CkStartCheckpoint("log_path", cb);

shell> ./charmrun hello +p4 +restart log_path
Control Flow within Chare

• Structured dagger notation
  • Provides a script-like language for expressing dag of dependencies between method invocations and computations

• Threaded entry methods
  • Allows entry methods to block without blocking the PE
  • Supports futures and
  • Ability to suspend/resume threads
Advanced Concepts

• Priorities
• Entry method tags
• Quiescence detection
• LiveViz: visualization from a parallel program
• CharmDebug: a powerful debugging tool
• Projections: Performance Analysis and Visualization, really nice, and a workhorse tool for Charm++ developers
• Messages (instead of marshalled parameters)
• Processor-aware constructs:
  • Groups: like a non-migratable chare array with one element on each “core”
  • Nodegroups: one element on each process
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
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
NAMD: Biomolecular Simulations

- Collaboration with K. Schulten
- With over 70,000 registered users
- Scaled to most top US supercomputers
- In production use on supercomputers and clusters and desktops
- Gordon Bell award in 2002

Determination of the structure of HIV capsid by researchers including Prof Schulten
Parallelization Using Charm++
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
OpenAtom:

On the fly ab initio molecular dynamics on the ground state surface with instantaneous GW-BSE level spectra

**PIs:** G.J. Martyna, IBM; S. Ismail-Beigi, Yale; L. Kale, UIUC;

**Team:** Q. Li, IBM, M. Kim, Yale; S. Mandal, Yale;
   E. Bohm, UIUC; N. Jain, UIUC; M. Robson, UIUC;
   E. Mikida, UIUC; P. Jindal, UIUC; T. Wicky, UIUC.
Decomposition and Computation Flow
## MiniApps

Available at: http://charmplusplus.org/miniApps/

<table>
<thead>
<tr>
<th>MiniApp</th>
<th>Features</th>
<th>Machine</th>
<th>Max cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMR</td>
<td>Over-decomposition, Custom array index, Message priorities, Load balancing, Checkpoint restart</td>
<td>BG/Q</td>
<td>131,072</td>
</tr>
<tr>
<td>LeanMD</td>
<td>Over-decomposition, 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>Over-decomposition, Message priorities, Load balancing</td>
<td>Blue Waters</td>
<td>16,384</td>
</tr>
<tr>
<td>LULESH 2.02</td>
<td>AMPI, Over-decomposition, Load balancing</td>
<td>Hopper</td>
<td>8,000</td>
</tr>
<tr>
<td>PDES</td>
<td>Over-decomposition, Message priorities, TRAM</td>
<td>Stampede</td>
<td>4,096</td>
</tr>
</tbody>
</table>
## More MiniApps

<table>
<thead>
<tr>
<th>MiniApp</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>
Describes seven major applications developed using Charm++

More info on Charm++:
http://charm.cs.illinois.edu
including the MiniApps
More information

• A lecture series with instructional material coming soon
  • http://charmplusplus.org/ (under “Learn”)
  • See also “Exercises” at the same link

• MiniApps (source code):
  • http://charmplusplus.org/miniApps/

• Research projects, papers, etc.
  • http://charm.cs.illinois.edu/

• Commercial support:
  • https://www.hpccharm.com/
Advanced Features and Application Case Studies

Brief peek at other features

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

© 2018 L. V. Kale at the University of Illinois Urbana
Control Flow within Chare

• Structured dagger notation
  • Provides a script-like language for expressing dag of dependencies between method invocations and computations

• Threaded entry methods
  • Allows entry methods to block without blocking the PE
  • Supports futures and
  • Ability to suspend/resume threads
Advanced Concepts

• Priorities
• Entry method tags
• Quiescence detection
• LiveViz: visualization from a parallel program
• CharmDebug: a powerful debugging tool
• Projections: Performance Analysis and Visualization, a workhorse tool for Charm++ developers
• Messages (instead of marshalled parameters)
• Processor-aware constructs:
  • Groups: like a non-migratable chare array with one element on each “core”
  • Nodegroups: one element on each process
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
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
NAMD: Biomolecular Simulations

- Collaboration with K. Schulten
- With over 70,000 registered users
- Scaled to most top US supercomputers
- In production use on supercomputers and clusters and desktops
- Gordon Bell award in 2002

Determination of the structure of HIV capsid by researchers including Prof Schulten
Parallelization Using Charm++

[Diagram of parallelization processes involving bond and non-bonded computations, patch integration, and point-to-point communication.]
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 K-D trees are considered better load balanced

Evolution of Universe and Galaxy Formation

With Charm++: use Oct-Tree, and let Charm++ map subtrees to processors
OpenAtom:

On the fly ab initio molecular dynamics on the ground state surface with instantaneous GW-BSE level spectra

**PIs:** G.J. Martyna, IBM; S. Ismail-Beigi, Yale; L. Kale, UIUC;

**Team:** Q. Li, IBM, M. Kim, Yale; S. Mandal, Yale;

- E. Bohm, UIUC; N. Jain, UIUC; M. Robson, UIUC;
- E. Mikida, UIUC; P. Jindal, UIUC; T. Wicky, UIUC.
Decomposition and Computation Flow
Describes seven major applications developed using Charm++

More info on Charm++:
http://charm.cs.illinois.edu
including the MiniApps
Available at: http://charmplusplus.org/miniApps/

<table>
<thead>
<tr>
<th>MiniApp</th>
<th>Features</th>
<th>Machine</th>
<th>Max cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMR</td>
<td>Over-decomposition, Custom array index, Message priorities, Load balancing, Checkpoint restart</td>
<td>BG/Q</td>
<td>131,072</td>
</tr>
<tr>
<td>LeanMD</td>
<td>Over-decomposition, 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>Over-decomposition, Message priorities, Load balancing</td>
<td>Blue Waters</td>
<td>16,384</td>
</tr>
<tr>
<td>LULESH 2.02</td>
<td>AMPI, Over-decomposition, Load balancing</td>
<td>Hopper</td>
<td>8,000</td>
</tr>
<tr>
<td>PDES</td>
<td>Over-decomposition, Message priorities, TRAM</td>
<td>Stampede</td>
<td>4,096</td>
</tr>
</tbody>
</table>
## More MiniApps

<table>
<thead>
<tr>
<th>MiniApp</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</td>
<td>65,536</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BG/Q</td>
<td>16,384</td>
</tr>
<tr>
<td>Random Access</td>
<td>TRAM</td>
<td>BG/P</td>
<td>131,072</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BG/Q</td>
<td>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>
More information

• A lecture series with instructional material coming soon
  • See also “Exercises” at the same link

• MiniApps (source code):
  • [http://charmplusplus.org/miniApps/](http://charmplusplus.org/miniApps/)

• Research projects, papers, etc.
  • [http://charm.cs.illinois.edu/](http://charm.cs.illinois.edu/)

• Commercial support:
  • [https://www.hpccharm.com/](https://www.hpccharm.com/)
Adaptive MPI
What Is Adaptive MPI?

• AMPI is an MPI implementation on top of Charm++’s runtime system
  • Enables Charm++’s dynamic features for pre-existing MPI codes

![Diagram showing MPI "Processes" implemented as virtual "processes" on Processor A and Processor B.](image)
Process Virtualization

• AMPI virtualizes MPI “ranks,” implementing them as migratable user-level threads rather than OS processes
  • Benefits:
    • Communication/computation overlap
    • Cache benefits to smaller working sets
    • Dynamic load balancing
    • Lower latency messaging within a process
  • Disadvantages:
    • Global/static variables are shared by all threads in an OS process scope
      • Not an issue for new applications
      • AMPI provides support for automating this at compile/runtime
      • Ongoing work to fully automate
Dynamic Load Balancing

• AMPI ranks are migratable across address spaces at runtime
  – Add a call to AMPI_Migrate(MPI_Info) in the application’s main iterative loop

• Isomalloc memory allocator
  • No need for the user to explicitly write de/serialization (PUP) routines
  • Memory allocator migrates all heap data and stack transparently
  • Works on all 64-bit platforms except BGQ & Windows
Fault Tolerance

• AMPI ranks can be migrated to persistent storage or in remote memories for fault tolerance
  • Storage can be Disk, SSD, NVRAM, etc.

• The runtime uses a scalable fault detection algorithm and restarts automatically on a failure
  • Restart is online, within the same job

• Checkpointing strategy is specified by passing a different *MPI_Info* to *AMPI_Migrate()*
Compiling & Running AMPI Programs

• To compile an AMPI program:
  • charm/bin/ampicc –o pgm pgm.o
  • For migratability, link with: -memory isomalloc
  • For LB strategies, link with: -module CommonLBs

• To run an AMPI job, specify the # of virtual processes (+vp)
  • ./charmrun +p 1024 ./pgm
  • ./charmrun +p 1024 ./pgm +vp 16384
  • ./charmrun +p 1024 ./pgm +vp 16384 +balancer RefineLB
Case Study

• LULESH proxy-application (LLNL)
  • Shock hydrodynamics on an unstructured mesh
  • With artificial load imbalance included to test runtimes

• No mutable global/static variables: can run on AMPI as is
  1. Replace mpicc with ampicc
  2. Link with “-module CommonLBs –memory isomalloc”
  3. Run with # of virtual processes and a load balancing strategy:
     • ./charmrun +p 2048 ./lulesh2.0 +vp 16384 +balancer GreedyLB
AMPI Summary

• AMPI provides the dynamic RTS support of Charm++ with the familiar API of MPI
  • Communication optimizations
  • Dynamic load balancing
  • Automatic fault tolerance
  • Checkpoint/restart
  • OpenMP runtime integration

• See the AMPI Manual for more info