Programming with Parallel Migratable Objects

Sanjay Kalé and PPL

Parallel Programming Laboratory
University of Illinois Urbana-Champaign

May 1, 2014

Manual: http://charm.cs.illinois.edu/manuals/html/charm++/manual.html

Installation: http://charm.cs.illinois.edu/manuals/html/charm++/A.html

Outline

Harnessing Parallelism: Challenges

Trends in System Architecture

- Frequencies have stopped increasing
- Memory costs are high
  - Relatively low per core memory
- Increasing heterogeneity
  - Accelerators, SMT
- Energy, power, and thermal considerations
- Frequent component failures

Harnessing Parallelism: Challenges

Trends in System Architecture

- However, compute resources are not faster cores, but **more cores**
- Unprecedented levels of available concurrency
  - IBM BG/Q
    - ‘Sequoia’: 1,572,864 cores
    - ‘Mira’: 786,432 cores
  - Cray
    - XE6+XK6 ‘Bluewaters‘: 386,816 cores
    - XK6 ‘Titan’: 299,008 cores
  - K Supercomputer: 705,024 cores
- Mid-size clusters will be ubiquitous

Harnessing Parallelism: Challenges

Trends in System Architecture

- However, compute resources are not faster cores, but **more cores**
- Unprecedented levels of available concurrency
  - IBM BG/Q
    - 'Sequoia': 1,572,864 cores
    - 'Mira': 786,432 cores
  - Cray
    - XE6+XK6 'Bluewaters': 386,816 cores
    - XK6 'Titan': 299,008 cores
  - K Supercomputer: 705,024 cores
- Mid-size clusters will be ubiquitous

Implications

- Each thread of execution has to:
  - operate on lesser data
  - wait relatively longer for remote data
- Have to operate in **strong scaling** regime

Harnessing Parallelism: Challenges

Next-generation Applications

- Need for strong scaling
  - faster solutions (not just larger problems)
- Application Characteristics
  - Multi-resolution
    - Adaptive, spatial and temporal resolutions
    - Dynamic/adaptive refinements: to handle application variation
  - Multi-module (multi-physics)
    - Complex physics in multiple, interacting modules
  - Adapt to a volatile computational environment
  - Exploit heterogeneous architecture
  - Deal with thermal and energy considerations

Harnessing Parallelism: Challenges

Next-generation Applications

- Need for strong scaling
  - faster solutions (not just larger problems)
- Application Characteristics
  - Multi-resolution
    - Adaptive, spatial and temporal resolutions
    - Dynamic/adaptive refinements: to handle application variation
  - Multi-module (multi-physics)
    - Complex physics in multiple, interacting modules
  - Adapt to a volatile computational environment
  - Exploit heterogeneous architecture
  - Deal with thermal and energy considerations
- So? Consequences:
  - Must support automated resource management
  - Must support interoperability and parallel composition

- Highly successful
- Universally used
- Has supported application evolution from gigascale to petascale

- Library
- Communication primitives

- MPI does not directly support automated resource management (e.g. load balancing, fault tolerance, etc.)

Charm++ builds upon a proven approach: objects

Stuff you already know

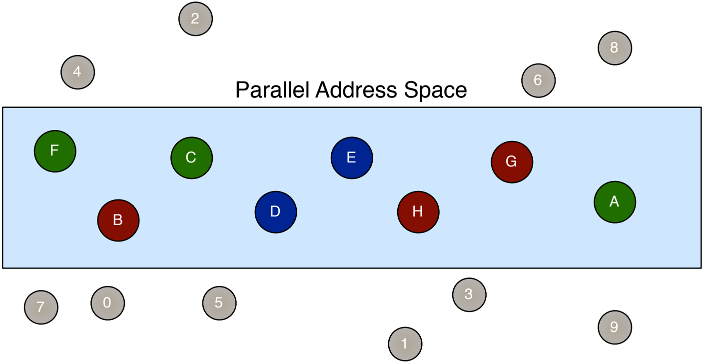
Benefits of Object-based code

- Objects encapsulate data
- Methods represent functionality relevant to that data
- Method invocations can modify / update state of the object / data
- Computation can be expressed in terms of objects interacting via method invocations

- Methods are natural units of sequential computation on object data
- Thoughtful design yields focused methods with single purpose
- Naturally expresses an object's response to inputs (signals / data)

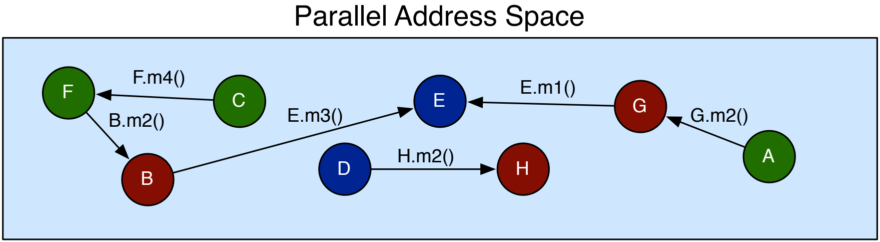
- Nothing new
- Still quite uncommon in HPC code
- Its not about language syntax. Its about program structure

Globally-Visible Objects: Chares and Proxies



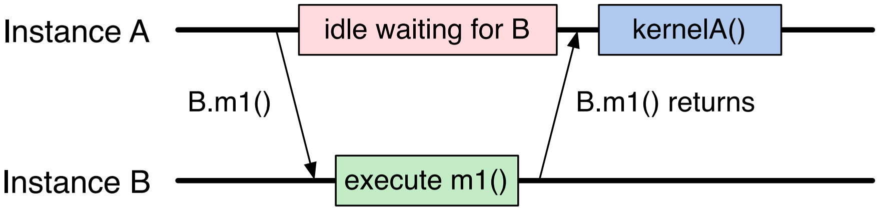
- Certain "special" object *instances* are:
  - first-class citizens in the parallel address space,
  - with unique location-independent names
- Under the hood, the runtime handles locality and provides the mechanisms to promote objects to the parallel space

Globally-Visible Methods: Entry Methods



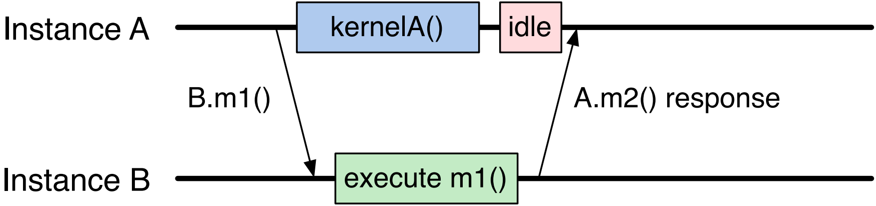
- How can objects communicate across address spaces?
  - Just like a sequential object-oriented language, an object's reference is used to invoke a method
  - In the parallel space, this is a handle that is location transparent
  - A method invocation becomes an act of communication

Method-Driven Asynchronous Communication



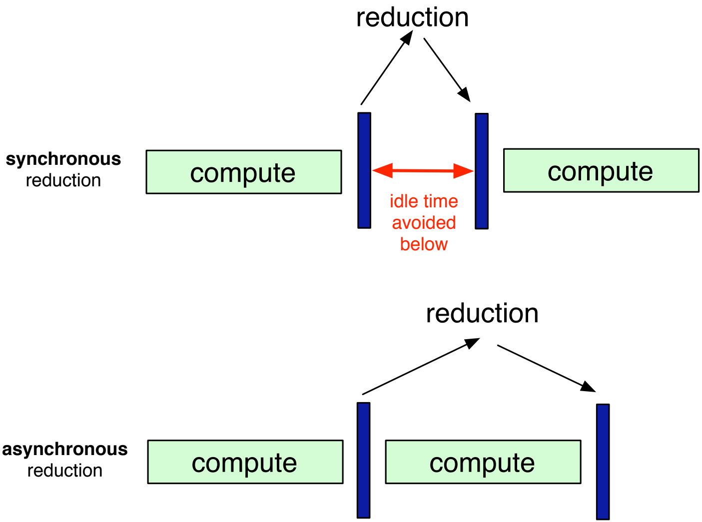
- What happens if an object waits for a return value from a method invocation?
  - Performance
  - Latency
  - Reasoning about correctness

Design Principle: Do not wait for remote completion



- Hence, method invocations should be asynchronous
  - No return values
- Computations are driven by the incoming data
  - Initiated by the sender or method caller

For example, a Jacobi reduction

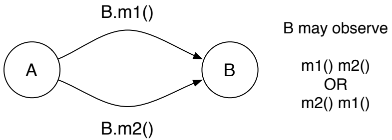


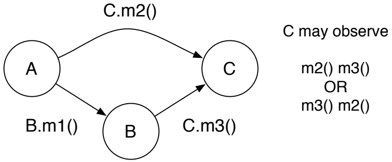
Methods: Natural Units of Sequential Computation

- Methods still have the same sequential semantics
  - Atomicity: methods do not execute in parallel
- Methods cannot be interrupted or preempted
- Methods interact and update state of an object in the same way
- Method sequencing is what changes from sequential computation

Methods: Natural Units of Sequential Computation

- Methods still have the same sequential semantics
  - Atomicity: methods do not execute in parallel
- Methods cannot be interrupted or preempted
- Methods interact and update state of an object in the same way
- Method sequencing is what changes from sequential computation





The Execution Model

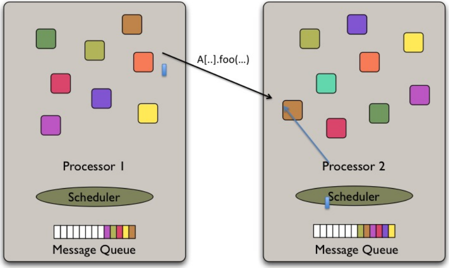
- Several objects live on a single *PE*
  - For now, think of it as a core (or just "processor")

The Execution Model

- Several objects live on a single *PE*
  - For now, think of it as a core (or just "processor")
- As a result,
  - Method invocations directed at objects on that processor will have to be stored in a pool,

The Execution Model

- Several objects live on a single *PE*
  - For now, think of it as a core (or just "processor")
- As a result,
  - Method invocations directed at objects on that processor will have to be stored in a pool,
  - And a user-level scheduler will select one invocation from the queue and runs it to completion
  - A PE is the entity that has one scheduler instance associated with it

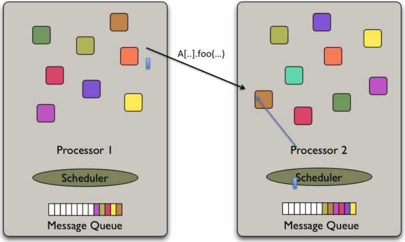


Message-driven Execution

- Execution is trigggered by availability of a "message" (a method invocation)

Message-driven Execution

- Execution is trigggered by availability of a "message" (a method invocation)
- When an entry method executes,
  - it may generate messages for other objects
  - the RTS deposits them in the message Q on the target processor



Outline

Hello World Example

- `hello.ci` file

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

- `hello.cpp` file

```
#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

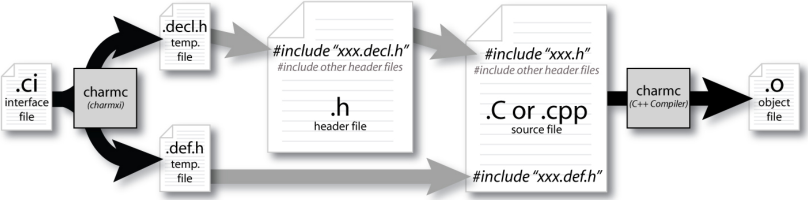
hello.ci file

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

hello.cpp file

```
#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() {
    ckout << "Hello World!" << endl;
    CkExit();
  };
};
#include "hello.def.h"
```

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.
- http://charm.cs.illinois.edu/manuals/html/charm++/A.html

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

Outline

Impact on communication

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

Impact on communication

- Current use of communication network
  - Compute-communicate cycles in typical MPI apps
  - Network is used for a fraction of time
  - And is on the critical path
- Hence, current communication networks are over-engineered by necessity

Impact on communication

- Current use of communication network
  - Compute-communicate cycles in typical MPI apps
  - Network is used for a fraction of time
  - And is on the critical path
- Hence, current communication networks are over-engineered by necessity
- With overdecomposition
  - Communication is spread over an iteration
  - Adaptive overlap of communication and computation

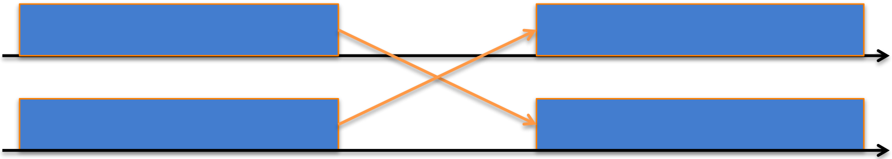
Example: Stencil Computation

- Consider a simple stencil computation
  - With traditional design based on traditional methods (e.g. MPI-based)
    - ★ Each processor has a chunk, which alternates between computing and communicating

Example: Stencil Computation

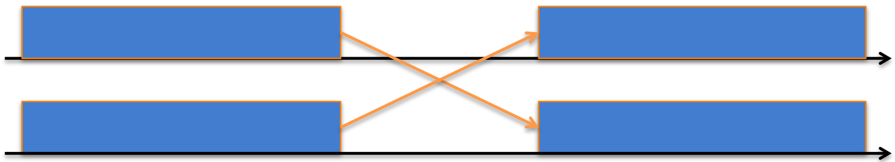
- Consider a simple stencil computation
  - With traditional design based on traditional methods (e.g. MPI-based)
    - Each processor has a chunk, which alternates between computing and communicating
  - With Charm++
    - Multiple chunks on each processor
    - Wait time for each chunk overlapped with useful computation for others
    - Communication spread over time

Example: Stencil Computation

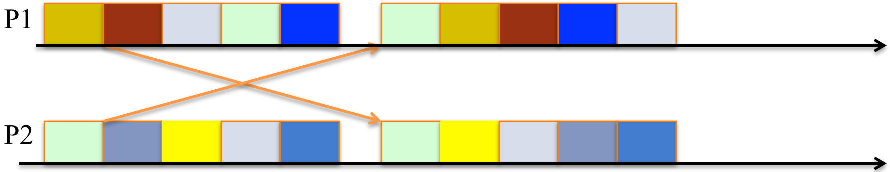


Stencil in MPI: No overlap among computation and communication

Example: Stencil Computation



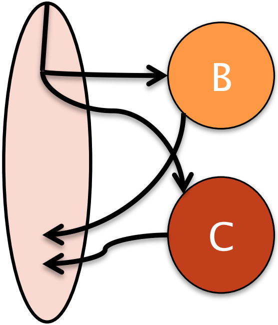
Stencil in MPI: No overlap among computation and communication

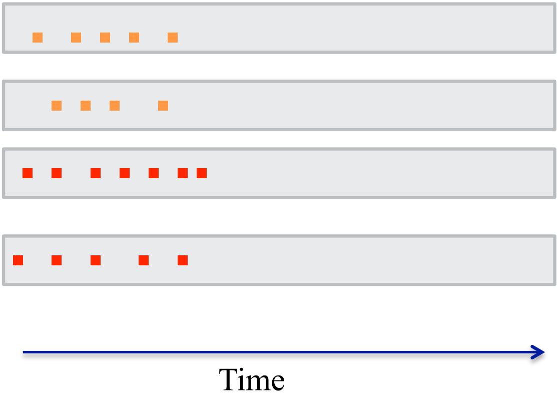


Stencil in Charm: Communication of a chare overlaps with computation of others

Modularity and Compositionality

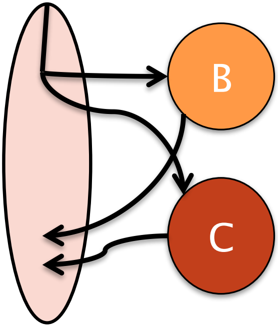
Without message-driven execution (and virtualization), you get either: Space-division

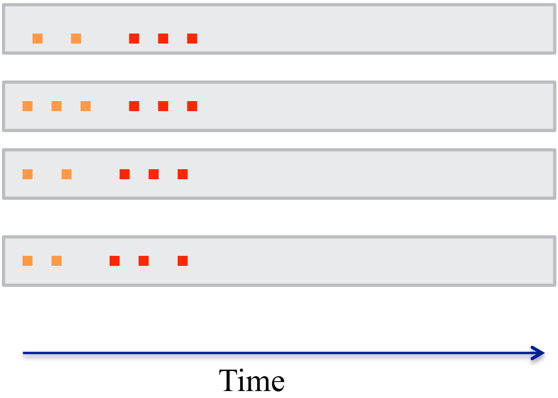




Modularity and Compositionality

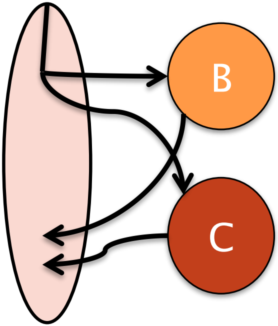
Sequentialization

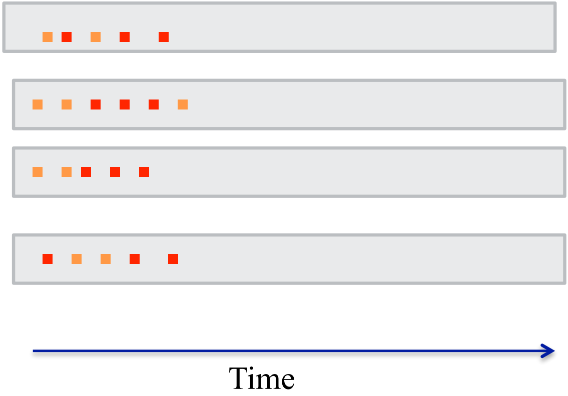




Modularity and Compositionality

Parallel Composition: A1; (B —— C); A2





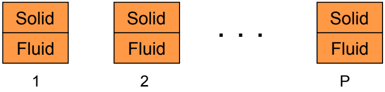
Recall: Different modules, written in different languages/paradigms, can overlap in time and on processors, without programmer having to worry about this explicitly

Migratability

- Once the programmer has written the code without reference to processors, all of the communication is expressed among objects
- The system is free to migrate the objects across processors as and when it pleases
  - It must ensure it can deliver method invocations to the objects, whereever they go
  - This migratability turns out to be a key attribute for empowering an adaptive runtime system

Decomposition Independent of numCores

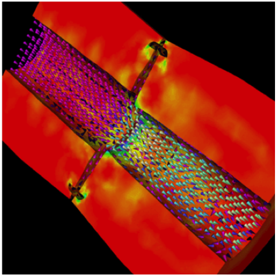
- Rocket simulation under traditional MPI



Decomposition Independent of numCores

- Rocket simulation under traditional MPI

Solid	Solid	· · ·	Solid
Fluid	Fluid		Fluid
1	2		P



- Rocket simulation with migratable objects

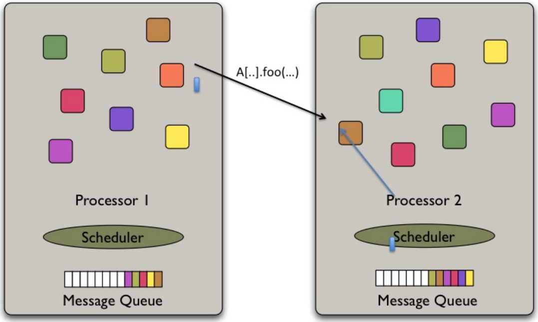
$Solid_1$ $Solid_2$ $Solid_3$ · · · $Solid_n$

$Fluid_1$ $Fluid_2$ · · · $Fluid_m$

  - Benefits: load balance, communication optimizations, modularity

Utility for Multi-cores, Many-cores, Accelerators

- Objects connote and promote locality
- Message-driven execution is
  - A strong principle of prediction for data and code use
  - Much stronger than principle of locality
    - ★ Can be used to scale memory wall
    - ★ Prefetching of needed data, e.g, into scratch pad memories

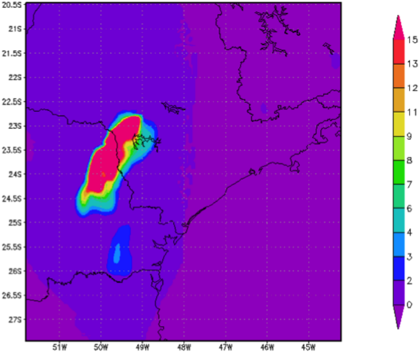


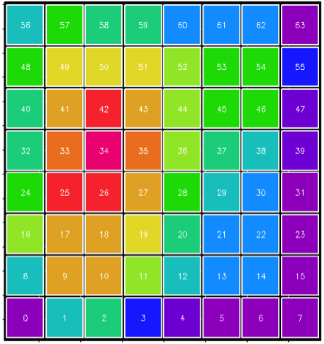
Load Balancing

- Static
  - Irregular applications
  - Programmer shouldn't have to figure out ideal mapping
- Dynamic
  - Applications are increasingly using adaptive strategies
  - Abrupt refinements
  - Continuous migration of work: e.g. particles in MD
- Challenges
  - Performance limited by most overloaded processor
  - The chance that one processor is severely overloaded gets higher as #processors increases

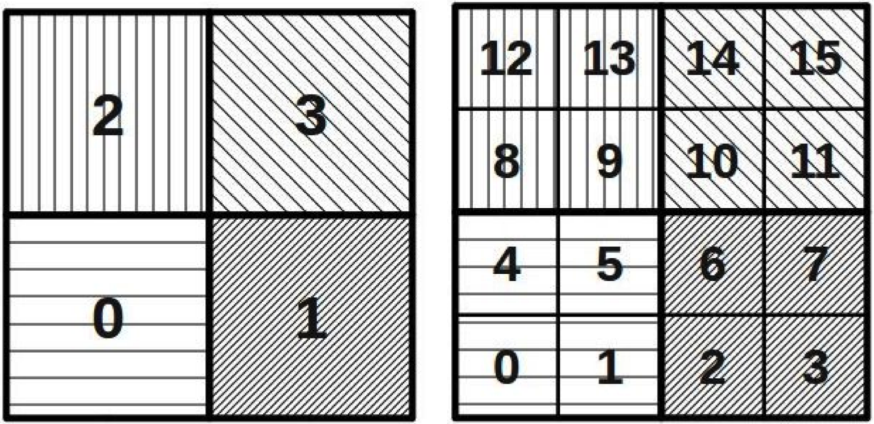
Migratable Objects Empower Automated Load Balancing!

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

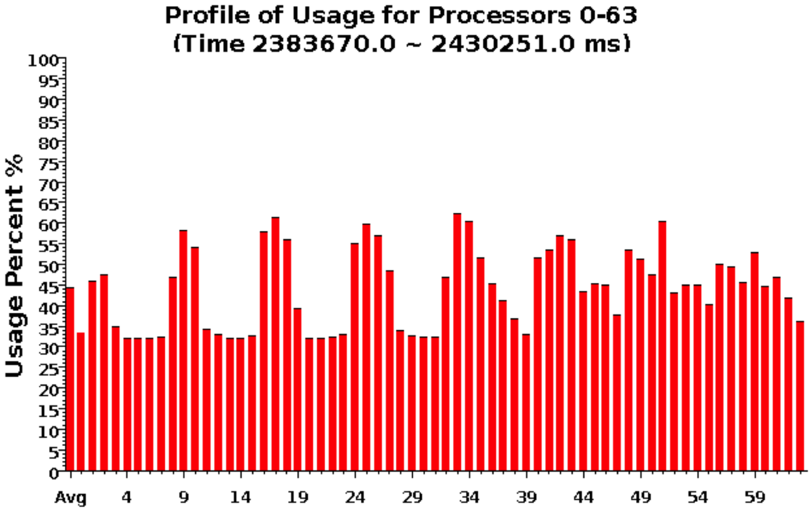




Basic Virtualization of BRAMS

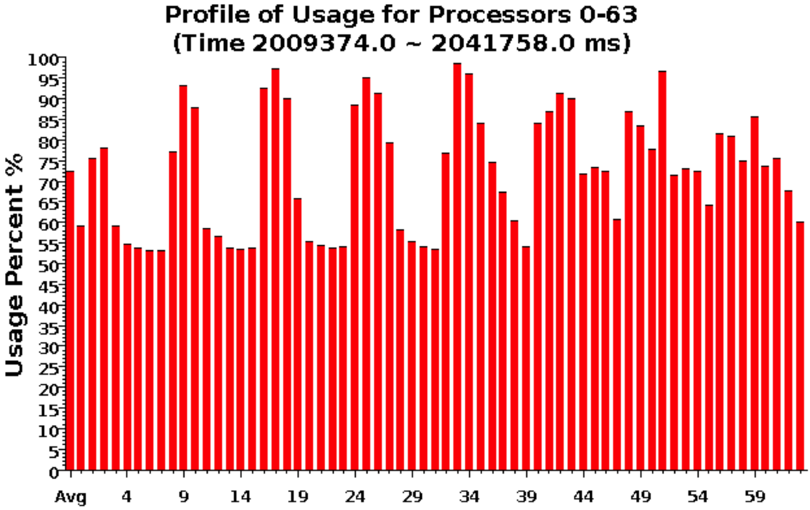


Baseline: 64 objects on 64 processors



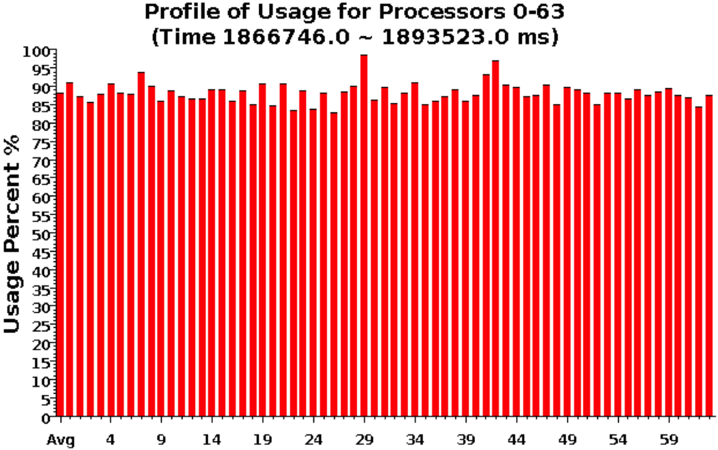
Over-decomposition: 1024 objects on 64 processors

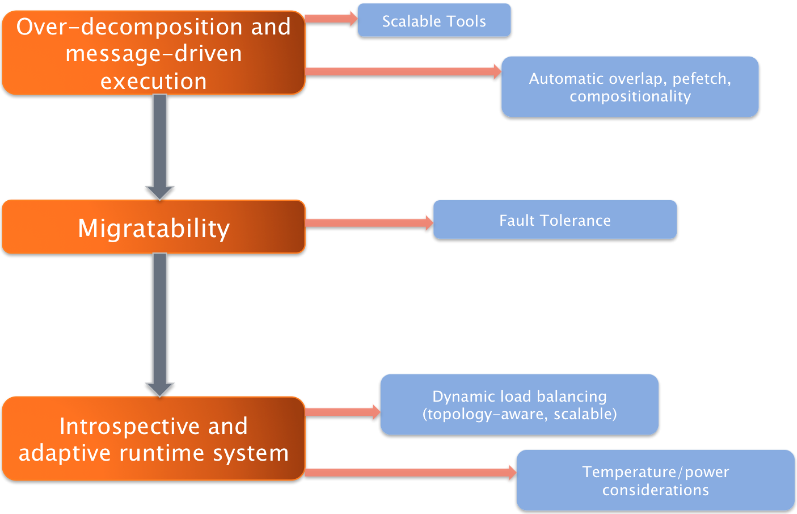
Benefits from communication/computation overlap



With Load Balancing: 1024 objects on 64 processors

- No overdecomp (64 threads): 4988 sec
- Overdecomp into 1024 threads: 3713 sec
- Load balancing (1024 threads): 3367 sec

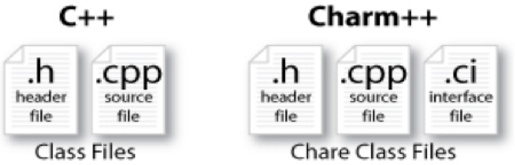



Over-decomposition and message-driven execution
Scalable Tools
Automatic overlap, pefetch, compositionality
Migratability
Fault Tolerance
Introspective and adaptive runtime system
Dynamic load balancing (topology-aware, scalable)
Temperature/power considerations

Outline

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 *mainchare*, is declared as the `mainmodule`
- Each module, when compiled, generates two files: `MyModule.decl.h` and `MyModule.def.h`

`.ci file`

```
[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`

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

`.C file`

```
class MyChare : public CBase_MyChare {
  //... entry method implementations ...
};
```

Charm Interface: Entry Methods

- Entry methods are C++ methods that can be remotely and asynchronously invoked by another chare

`.ci` file:

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

`.C` file:

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

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

MyChare::bar(int param) { /*... code to execute ...*/ }
```

Charm Interface: `mainchare`

- Execution begins with the `mainchare`'s constructor
- The `mainchare`'s constructor takes a pointer to system-defined class `CkArgMsg`
- `CkArgMsg` contains `argv` and `argc`
- The `mainchare` will typically creates some additional chares

Creating a Chare

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

```
CProxy_MyChare::ckNew(... constructor arguments ...);
```

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

```
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

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

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

Chare Creation Example: .C file

```
#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

```
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:

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

- These prints may occur in **any** order

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

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

Asynchronous Methods

- For example, if a chare invokes the same entry method twice:

```
proxy.bar(7);
proxy.bar(5);
```

- These may be delivered in **any** order

```
MyChare::bar(int param) {
  ckout << "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?

```
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;
   ckout << "Hello from a simple chare running on " << CkMyPe() << endl;
 }
 void findArea(int r, bool done) {
   ckout << "Area of a circle of radius" << r << " is " << y*r*r << endl;
   if (done) CkExit();
 }
```

Data types and entry methods

- 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:

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

- `.C` file:

```
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

Chare Array: Hello Example

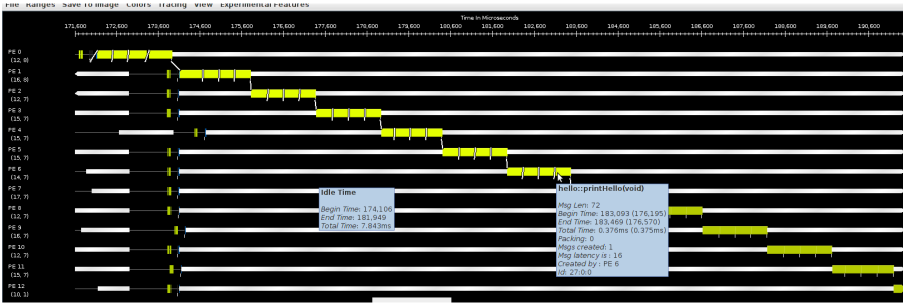
```
mainmodule arr {

  mainchare Main {
    entry Main(CkArgMsg*);
  }

  array [1D] hello {
    entry hello(int);
    entry void printHello();
  }
}
```

Chare Array: Hello Example

```
#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:

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

.C file:

```
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

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

- The proxy may be retained:

```
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

```
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:

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

`.C` file:

```
struct foo : public CBase_foo {
  foo() {
    CkPrintf("array index = %d", thisIndex);
  }
};
```

Chare Array: Hello Example

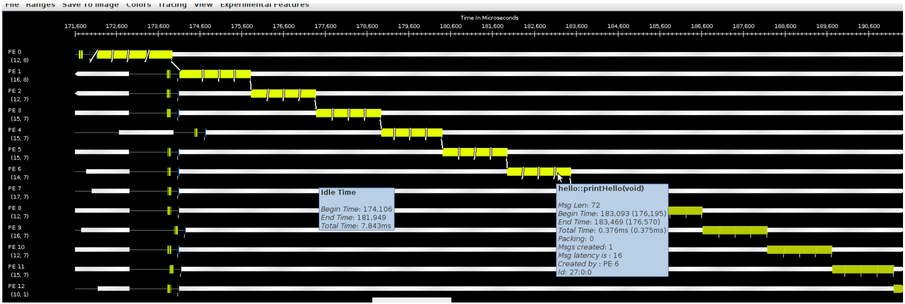
```
mainmodule arr {

  mainchare Main {
    entry Main(CkArgMsg∗);
  }

  array [1D] hello {
    entry hello(int);
    entry void printHello();
  }
}
```

Chare Array: Hello Example

```
#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)

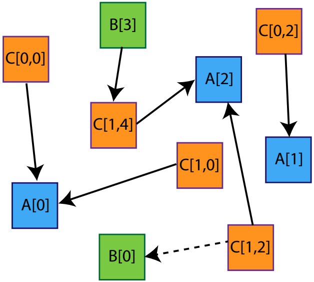
Collections of Objects: Runtime Service

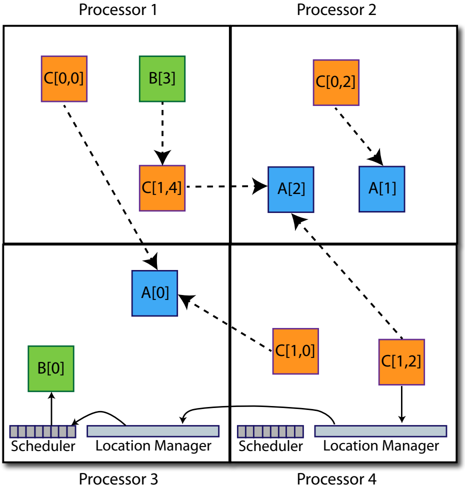
- System knows how to ‘find’ objects efficiently: $(collection, index) \rightarrow processor$
- Applications can specify a mapping, or use simple runtime-provided options (e.g. blocked, round-robin)
- Distribution can be static, or dynamic!
- Key abstraction: application logic doesn’t change, even though 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

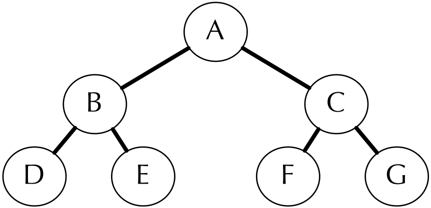
Collections of Objects





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



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:

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

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

```
thisProxy.foo()
```

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

```
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

```
mainmodule reduction {
  mainchare Main {
    entry Main(CkArgMsg* msg);
    entry [reductiontarget] void done(int value);
  };
  array [1D] Elem {
    entry Elem(CProxy_Main mProxy);
  };
}
```

Reduction: Example

```
#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.
```

Outline

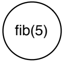
Task Parallelism with Objects

- Divide-and-conquer
  - Each object recursively creates n objects that divide the problem into subproblems
  - Each object t then waits for all n objects to finish and then may 'combine' the responses
  - At some point the recursion stops (at the bottom of the tree), and some sequential kernel is executed
  - Then the result is propagated upward in the tree recursively
  - Examples: fibonacci, quick sort, ...

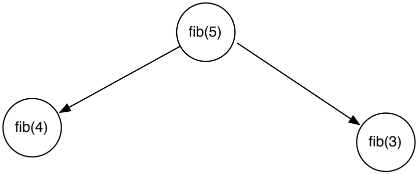
Fibonacci Example

- Each `Fib` object is a task that performs one of two actions:
  - Creates two new `Fib` objects to compute $fib(n-1)$ and $fib(n-2)$ and then waits for the response, adding up the two responses when they arrive
    - After both arrive, sends a response message with the result to the parent object
    - Or prints the value and exits if it is the root
  - If $n = 1$ or $n = 0$ (passed down from the parent) it sends a response message with n back to the parent object

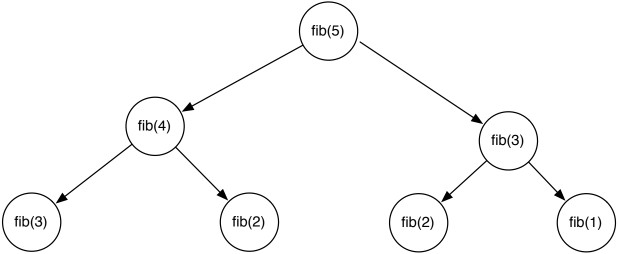
Fibonacci Execution



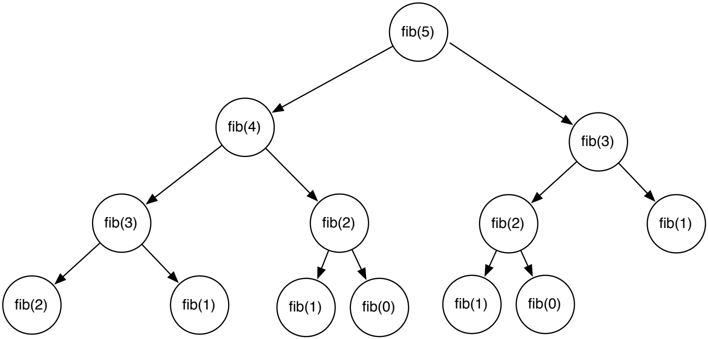
Fibonacci Execution



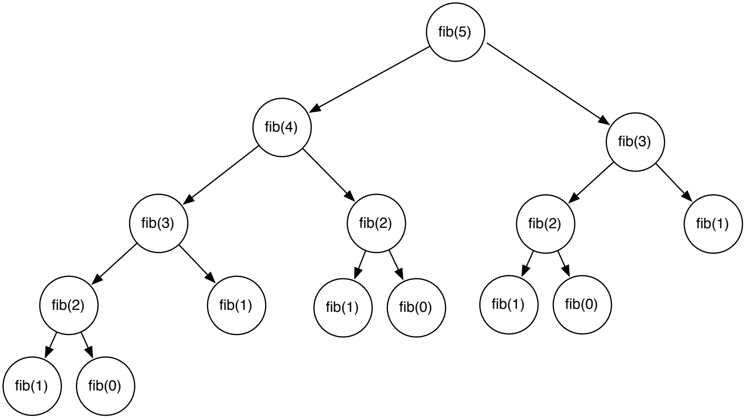
Fibonacci Execution



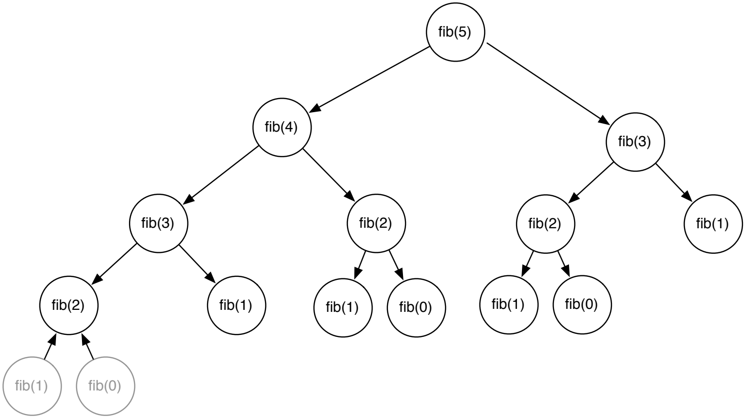
Fibonacci Execution



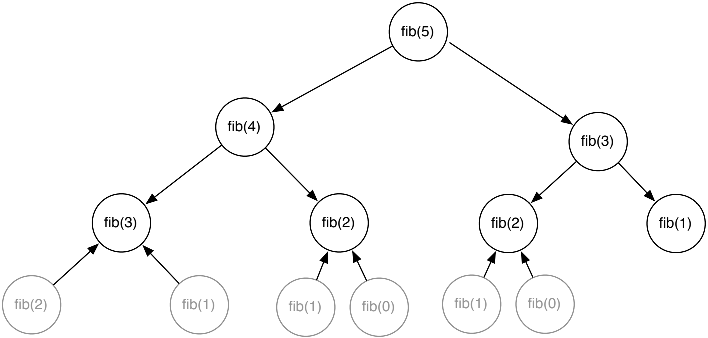
Fibonacci Execution



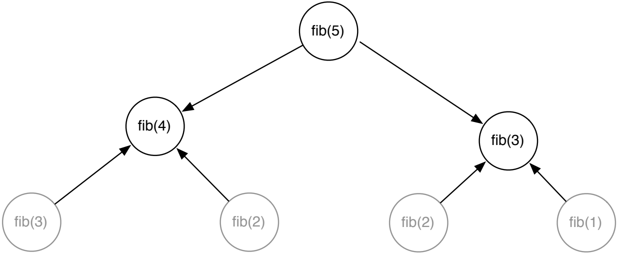
Fibonacci Execution



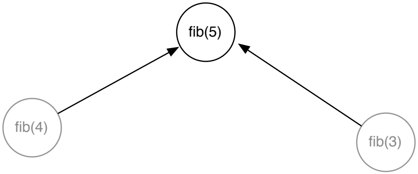
Fibonacci Execution



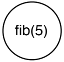
Fibonacci Execution



Fibonacci Execution



Fibonacci Execution



Overdecomposing Your Application

Object-based Over-decomposition

- Let the programmer decompose computation into objects
  - Work units, data-units, composites
- Let an intelligent runtime system assign objects to processors
  - RTS can change this assignment (mapping) during execution
  - Locality of data references is a critical attribute for performance
  - A parallel object can access only its own data
  - Asynchronous method invocation for accessing other objects data
  - RTS can schedule work whose dependencies have been satisfied

Amdahls Law and Grainsize

- Original "law":
  - If a program has $K\%$ sequential section, then speedup is limited to $\frac{100}{K}$.
    - ★ If the rest of the program is parallelized completely
- Grainsize corollary:
  - If any individual piece of work is $> K$ time units, and the sequential program takes T_{seq},
    - ★ Speedup is limited to $\frac{T_{seq}}{K}$
- So:
  - Examine performance data via histograms to find the sizes of remappable work units
  - If some are too big, change the decomposition method to make smaller units

Overdecomposition and Grainsize

- Common misconception: overdecomposition must be expensive
- (working) Definition: the amount of computation per potentially parallel event (task creation, enqueue/dequeue, messaging, locking, etc.)

Grainsize and Overhead

- What is the ideal grainsize?
- Should it depend on the number of processors?

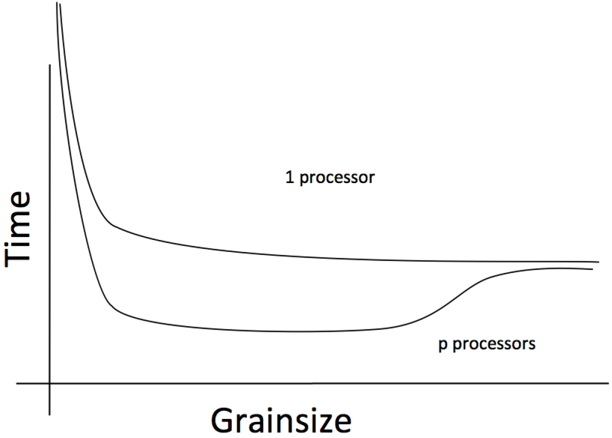
$$T_1 = T\left(1 + \frac{v}{g}\right)$$
$$T_p = max\left\{g, \frac{T_1}{p}\right\}$$
$$T_p = max\left\{g, \frac{T\left(1+\frac{v}{g}\right)}{p}\right\}$$

v: overhead per message,

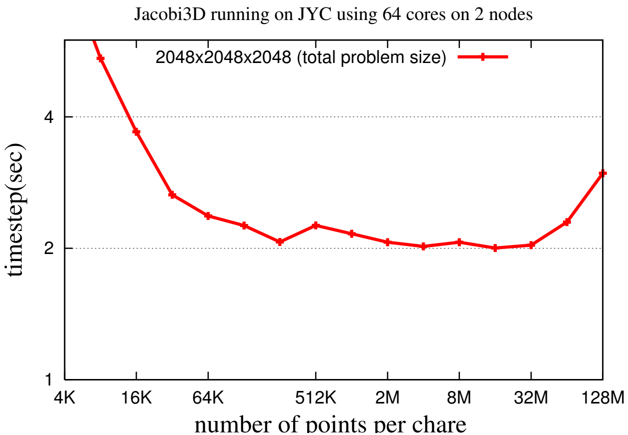
T_p: p processor completion time

g: grainsize (computation per message)

Grainsize and Scalability

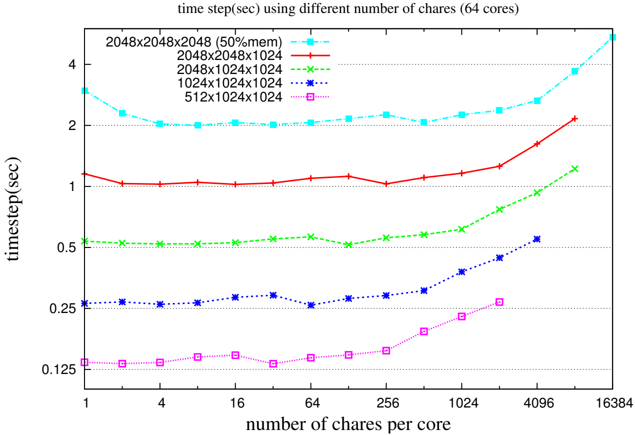


Grainsize Study for Jacobi3D



Grainsize Study for Stencil Computation

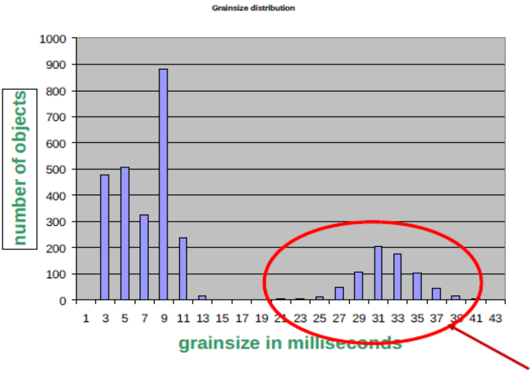
- Blue Waters (JYC) , 2 nodes, 32 cores each



Typically, having tens of chares per code is adequate (although reasoning should be based on computation per message)

Grainsize and Load Balancing

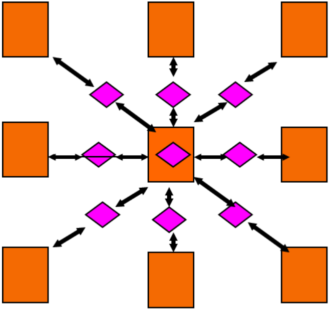
How Much Balance Is Possible?

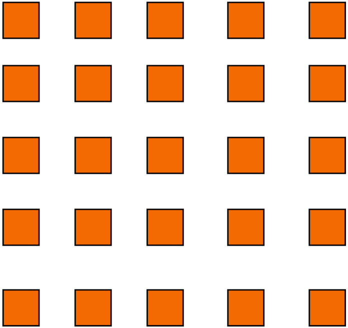


Solution:
Split compute objects that may have too much work, using a heuristic based on number of interacting atoms

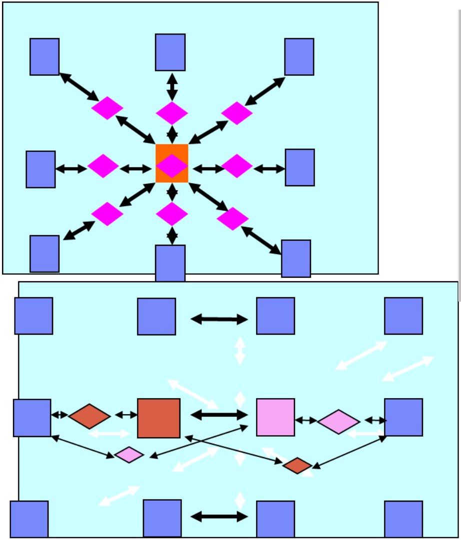
Grainsize For Extreme Scaling

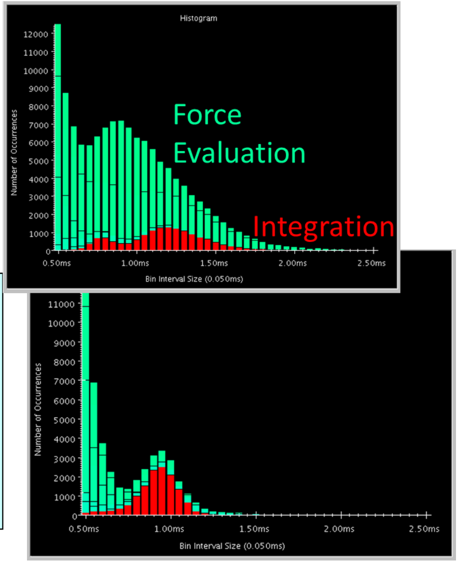
- Strong Scaling is limited by expressed parallelism
  - Minimum iteration time limited by lengthiest computation
    - Largest grains set lower bound
- 1-away generalized to k-away provides fine granularity control





NAMD: 2-AwayX Example



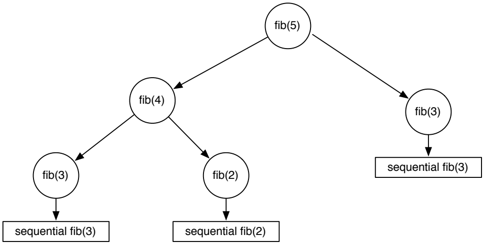


Rules of thumb for grainsize

- Make it as small as possible, as long as it amortizes the overhead
- More specifically, ensure:
  - *Average* grainsize is greater than kv (say $10v$)
  - No single grain should be allowed to be too large
    - ★ Must be smaller than $\frac{T}{p}$, but actually we can express it as:
    - ★ Must be smaller than kmv (say $100v$)
- Important corollary:
  - You can be at close to optimal grainsize without having to think about p, the number of processors
- $kv < g < mkv \; (10v < g < 100v)$

Grain size for Fibonacci Example

- Set a sequential threshold in the computational tree
  - Past this threshold (i.e. when $n < threshold$), instead of constructing two new chares, compute the fibonacci sequentially

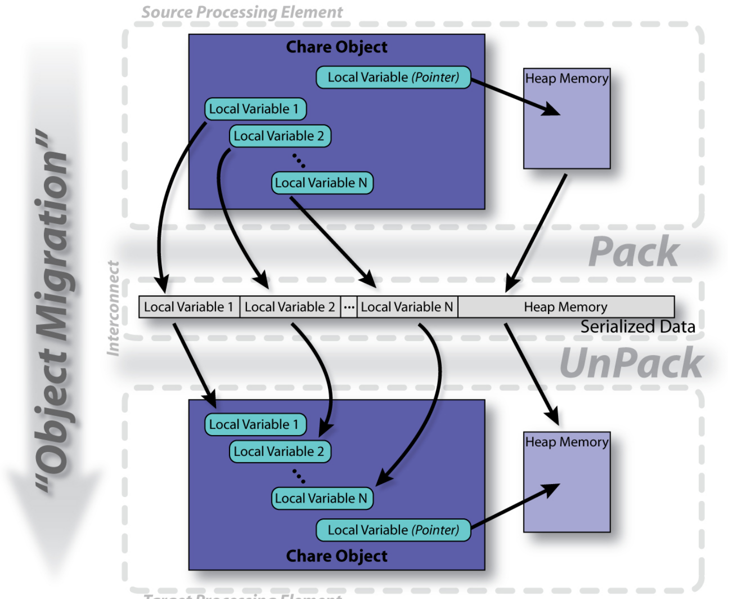


- Setting the grainsize limit at 4 (which is too small, but good for illustration)
- The internal nodes of the tree do very little work, but
- The coarser grains now amortize the cost of the fine-grained chares

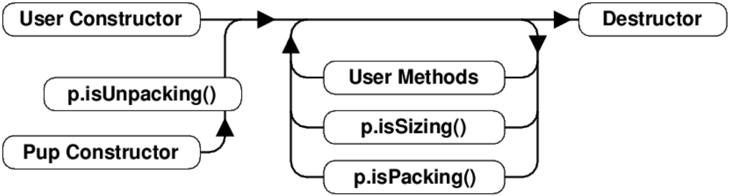
Outline

Object Serialization Using PUP: The Pack/UnPack Framework

The PUP Process



PUP Usage Sequence



- Migration out:
  - ckAboutToMigrate
  - Sizing
  - Packing
  - Destructor
- Migration in:
  - Migration constructor
  - UnPacking
  - ckJustMigrated

Writing a PUP routine

```
class MyChare : public
      CBase_MyChare {
   int a;
   float b;
   char c;
   float localArray[LOCAL_SIZE];
};
```

```
void pup(PUP::er &p) {
    CBase_MyChare::pup(p);
    p | a;
    p | b;
    p | c;
    p(localArray, LOCAL_SIZE);
}
```

Writing a PUP routine

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

```
void pup(PUP::er &p) {
    CBase_MyChare::pup(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: Concerns

- If variables are added to an object, update the PUP routine
- If the object allocates data on the heap, copy it recursively, not just the pointer
- Remember to allocate memory while unpacking
- Sizing, Packing, and Unpacking must scan the variables in the same order
- Test PUP routines with +balancer RotateLB

Fault Tolerance in Charm++/AMPI

- Four Approaches:
  - Disk-based checkpoint/restart
  - In-memory double checkpoint/restart
  - Experimental: Proactive object evacuation
  - Experimental: Message-logging for scalable fault tolerance
- Common Features:
  - Easy checkpoint
  - Migrate-to-disk leverages object-migration capabilities
  - Based on dynamic runtime capabilities
  - Can be used in concert with load-balancing schemes

Checkpointing to the file system : Split Execution

- The common form of checkpointing
  - The job runs for 5 hours, then will continue at the next allocation another day!
- The existing Charm++ infrastructure for chare migration helps
- Just "migrate" chares to disk
- The call to checkpoint the application is made in the main chare at a synchronization point

```
    CkCallback cb(CkIndex_Hello::SayHi(),helloProxy);
    CkStartCheckpoint(''log'',cb);

> ./charmrun hello +p4 +restart log
```

Outline

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)!

Fibonacci Example

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

  chare Fib {
    entry Fib(int n, bool isRoot, CProxy_Fib parent);
    entry void respond(int value);
  };
};
```

Fibonacci Example

```
class Main : public CBase_Main {
public: Main(CkArgMsg* m) {
    CProxy_Fib::ckNew(atoi(m->argv[1]), true, CProxy_Fib());
  }
};

class Fib : public CBase_Fib {
public: CProxy_Fib parent; bool isRoot; int result, count;

  Fib(int n, bool isRoot_, CProxy_Fib parent_)
    : parent(parent_), isRoot(isRoot_), result(0), count(2) {

    if (n < 2) respond(n);
    else {
      CProxy_Fib::ckNew(n - 1, false, thisProxy);
      CProxy_Fib::ckNew(n - 2, false, thisProxy);
    }
  }

  void respond(int val) {
    result += val;
    if (--count == 0 || n < 2) {
      if (isRoot) {
        CkPrintf("Fibonacci number is: %d\n", result);
        CkExit();
      } else {
        parent.respond(result);
        delete this;
      }
    }
  }
};
```

Consider Fibonacci Chare

- The Fibonacci chare gets created
- If its not a leaf,
  - It fires two chares
  - When both children return results (by calling `respond`):
    - It can compute my result and send it up, or print it
  - But in our, this logic is hidden in the flags and counters ...
    - This is simple for this simple example, but ...
  - Lets look at how this would look with a little notational support

- The `when` construct
  - Declare the actions to perform when a message is received
  - In sequence, it acts like a blocking receive

```
entry void someMethod() {
  when entryMethod1(parameters) { /* block2 */ }
  when entryMethod2(parameters) { /* 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, like an entry method
  - Syntax: `serial <optionalString> { /* C++ code */ }`
  - The `<optionalString>` is used for identifying the `serial` for performance analysis
  - Serial blocks can access all members of the class they belong to
- Examples (.ci file):

```
entry void method1(parameters) {
  serial {
    thisProxy.invokeMethod(10);
    callSomeFunction();
  }
};
```

```
entry void method2(parameters) {
  serial "setValue" {
    value = 10;
  }
};
```

Structured Dagger

The `when` construct

```
entry void someMethod() {
  serial { /* block1 */ }
  when entryMethod1(parameters) serial { /* block2 */ }
  when entryMethod2(parameters) serial { /* block3 */ }
};
```

- Sequence

Structured Dagger

The `when` construct

```
entry void someMethod() {
  serial { /* block1 */ }
  when entryMethod1(parameters) serial { /* block2 */ }
  when entryMethod2(parameters) serial { /* block3 */ }
};
```

- Sequence
  - Sequentially execute `/* block1 */`

```
entry void someMethod() {
  serial { /* block1 */ }
  when entryMethod1(parameters) serial { /* block2 */ }
  when entryMethod2(parameters) serial { /* block3 */ }
};
```

- Sequence
  - Sequentially execute `/* block1 */`
  - Wait for `entryMethod1` to arrive, if it has not, return control back to the Charm++ scheduler, otherwise, execute `/* block2 */`

Structured Dagger

The `when` construct

```
entry void someMethod() {
  serial { /* block1 */ }
  when entryMethod1(parameters) serial { /* block2 */ }
  when entryMethod2(parameters) serial { /* block3 */ }
};
```

- Sequence
  - Sequentially execute `/* block1 */`
  - Wait for `entryMethod1` to arrive, if it has not, return control back to the Charm++ scheduler, otherwise, execute `/* block2 */`
  - Wait for `entryMethod2` to arrive, if it has not, return control back to the Charm++ scheduler, otherwise, execute `/* block3 */`

Structured Dagger

The `when` construct

- Execute `/* further sdag */` when `myMethod` arrives

```
when myMethod(int param1, int param2)
  /* further code */
```

- Execute `/* further sdag */` when `myMethod1` and `myMethod2` arrive

```
when myMethod1(int param1, int param2),
       myMethod2(bool param3)
  /* further code */
```

- Which is almost the same as this:

```
when myMethod1(int param1, int param2) {
  when myMethod2(bool param3) { }
}
    /* further code */
```

Structured Dagger

Boilerplate

- Structured Dagger can be used in any entry method (except for a constructor)
  - Can be used in a `mainchare`, `chare`, or `array`
- For any class that has Structured Dagger in it you must insert two calls:
  - The Structured Dagger macro: `[ClassName]_SDAG_CODE`
  - For later: call the `__sdag_pup()` in the `pup` method

Structured Dagger

Boilerplate

The .ci file:

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

The .cpp file:

```
class MyFoo : public CBase_MyFoo {
  MyFoo_SDAG_CODE /* insert SDAG macro */
public:
  MyFoo() { }
};
```

Fibonacci with Structured Dagger

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

  chare Fib {
    entry Fib(int n, bool isRoot, CProxy_Fib parent);
    entry void calc(int n) {
      if (n < THRESHOLD) serial { respond(seqFib(n)); }
      else {
        serial {
          CProxy_Fib::ckNew(n - 1, false, thisProxy);
          CProxy_Fib::ckNew(n - 2, false, thisProxy);
        }
        when response(int val)
          when response(int val2)
            serial { respond(val + val2); }
      }
    };
    entry void response(int);
  };
};
```

Fibonacci with Structured Dagger

```
#include "fib.decl.h"
#define THRESHOLD 10

class Main : public CBase_Main {
public: Main(CkArgMsg* m) { CProxy_Fib::ckNew(atoi(m->argv[1]), true, CProxy_Fib()); }
};

class Fib : public CBase_Fib {
public:
  Fib_SDAG_CODE
  CProxy_Fib parent; bool isRoot;

  Fib(int n, bool isRoot_, CProxy_Fib parent_)
    : parent(parent_), isRoot(isRoot_) {
    calc(n);
  }

  int seqFib(int n) { return (n < 2) ? n : seqFib(n - 1) + seqFib(n - 2); }

  void respond(int val) {
    if (!isRoot) {
      parent.response(val);
      delete this;
    } else {
      CkPrintf("Fibonacci number is: %d\n", val);
      CkExit();
    }
  }
};

#include "fib.def.h"
```

Structured Dagger

The when construct

- What is the sequence?

```
when myMethod1(int param1, int param2) {
  when myMethod2(bool param3),
       myMethod3(int size, int arr[size]) /* sdag block1 */
  when myMethod4(bool param4) /* sdag block2 */
}
```

Structured Dagger

The `when` construct

- What is the sequence?

```
when myMethod1(int param1, int param2) {
  when myMethod2(bool param3),
       myMethod3(int size, int arr[size]) /* sdag block1 */
  when myMethod4(bool param4) /* sdag block2 */
}
```

- Sequence:
  - Wait for `myMethod1`, upon arrival execute body of `myMethod1`

- What is the sequence?

```
when myMethod1(int param1, int param2) {
  when myMethod2(bool param3),
       myMethod3(int size, int arr[size]) /* sdag block1 */
  when myMethod4(bool param4) /* sdag block2 */
}
```

- Sequence:
  - Wait for `myMethod1`, upon arrival execute body of `myMethod1`
  - Wait for `myMethod2` and `myMethod3`, upon arrival of both, execute `/* sdag block1 */`

Structured Dagger

The `when` construct

- What is the sequence?

```
when myMethod1(int param1, int param2) {
  when myMethod2(bool param3),
        myMethod3(int size, int arr[size]) /* sdag block1 */
  when myMethod4(bool param4) /* sdag block2 */
}
```

- Sequence:
  - Wait for `myMethod1`, upon arrival execute body of `myMethod1`
  - Wait for `myMethod2` and `myMethod3`, upon arrival of both, execute `/* sdag block1 */`
  - Wait for `myMethod4`, upon arrival execute `/* sdag block2 */`
- Question: if `myMethod4` arrives first what will happen?

Structured Dagger Constructs

The `when` construct

- The `when` clause can wait on a certain reference number
- If a reference number is specified for a `when`, the first parameter for the `when` must be the reference number
- Semantic: the `when` will "block" until a message arrives with that reference number

```
when method1[100](int ref, bool param1)
  /* sdag block */

serial {
  proxy.method1(200, false); /* will not be delivered to the when */
  proxy.method1(100, true); /* will be delivered to the when */
}
```

- The `if-then-else` construct:
  - Same as the typical C if-then-else semantics and syntax

```
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 ith iteration completes, the $i + 1$ iteration is started

```
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

```
class Foo : public CBase_Foo {
  public: int iter;
};
```

- The `while` construct:
  - Defines a sequenced `while` loop (like a sequential C while loop)

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

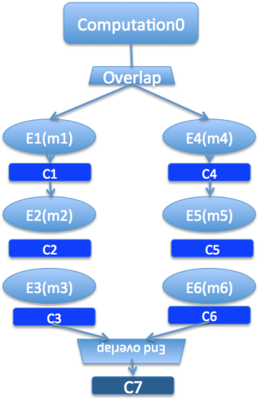
- The `overlap` construct:
  - By default, Structured Dagger defines a sequence that is followed sequentially
  - `overlap` allows multiple independent clauses to execute in any order
  - Any constructs in the body of an `overlap` can happen in any order
  - An `overlap` finishes in sequence when all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

What are the possible execution sequences?

```
serial { /* block1 */ }
overlap {
  serial { /* block2 */ }
  when entryMethod1[100](int ref_num, bool param1) /* block3 */
  when entryMethod2(char myChar) /* block4 */
}
serial { /* block5 */ }
```

Illustration of a long "overlap"

- Overlap can be used to get back some of the asynchrony within a chare
  - But it is constrained
  - Makes for more disciplined programming,
    - with fewer race conditions



Structured Dagger

The `forall` construct

- The `forall` construct:
  - Has "do-all" semantics: iterations may execute an any order
  - Syntax:
    `forall [<ident>] (<min> : <max>, <stride>) <body>`
  - The range from `<min>` to `<max>` is inclusive

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

- Assume `block` is declared in the class as `public: int block;`

Parallel Prefix with SDAG: .ci file I

```
mainmodule prefix {



  mainchare Main {
    entry Main(CkArgMsg* msg);
    entry [reductiontarget] void checkIn();
  };


  array [1D] Prefix {
    entry Prefix(int n, CProxy_Main m);
    entry void passValue(int step, unsigned int incomingValue);
```

Parallel Prefix with SDAG: .ci file II

```
    entry void startPrefixCalculation() {
      for(stage = 0; (1 << stage) < numElements; stage++) {
        serial "send_value" {
          targetIndex = thisIndex + (1<<stage);
          if (targetIndex < numElements)
            thisProxy[targetIndex].passValue(stage, value);
        }
        if (thisIndex >= (1<<stage))
          when passValue[stage] (int incoming_stage, unsigned int incoming_value) serial
               {
            value += incoming_value;
          }
      }
      serial "done" {
        contribute(CkCallback(CkReductionTarget(Main, checkIn), mainProxy));
      }
    };
  };
};
```

Parallel Prefix with SDAG: .C file I

```
#include "prefix.decl.h"
#include <stdlib.h>

class Main : public CBase_Main {
  public:
  CProxy_Prefix prefixArray;

  Main(CkArgMsg* msg) {
    int numElements = 10;
    if (msg->argc > 1)
      numElements = atoi(msg->argv[1]);
    prefixArray = CProxy_Prefix::ckNew(numElements, thisProxy, numElements);
    prefixArray.startPrefixCalculation();
  }
  Main(CkMigrateMessage* msg) { }

  void checkIn() {
    CkExit();
  }
};
```

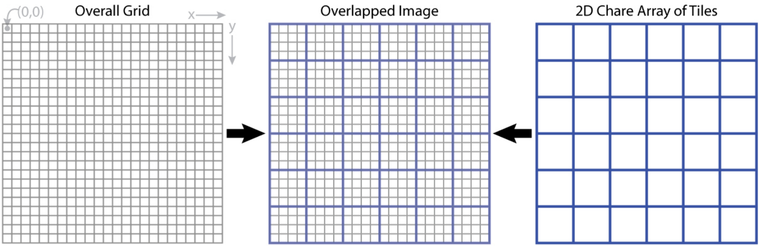
Parallel Prefix with SDAG: .C file II

```
class Prefix : public CBase_Prefix {
  Prefix_SDAG_CODE

  public:
    int stage, targetIndex, value, numElements;
    CProxy_Main mainProxy;

    Prefix(int n, CProxy_Main p) : numElements(n), mainProxy(p) {
      srand(thisIndex);
      value = rand() % 10; // Random positive int between 0 and 9 (inclusive)
    }
    Prefix(CkMigrateMessage *msg) { }

};
#include "prefix.def.h"
```

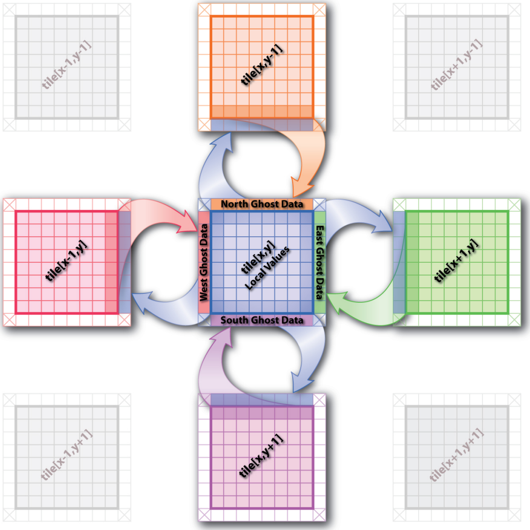
Stencil Codes

- Iterative applications where array elements are updated according to some fixed pattern.
- Used in computational simulations, solving partial differential equations, Jacobi kernel, GaussSeidel method, image processing applications etc.
- Can be 2D or 3D

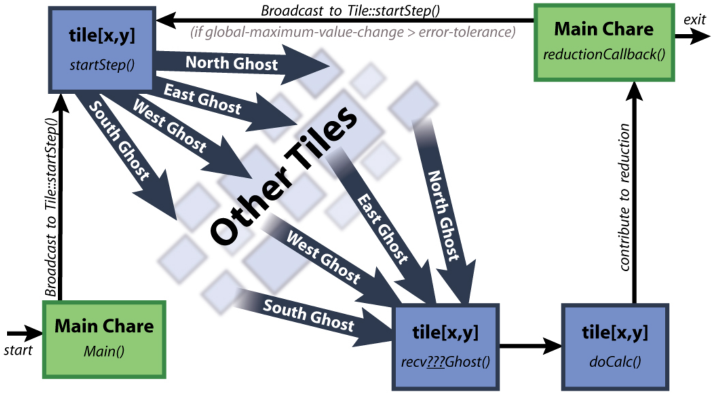
5-point Stencil



5-point Stencil



5-point Stencil



Jacobi: .ci file

```
mainmodule jacobi3d {

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

  array [3D] Jacobi {
    entry Jacobi(CProxy_Main);
    entry void updateGhosts(int ref, int dir, int w, int h, double gh[w*h]);
    entry [reductiontarget] void checkConverged(bool result);
    entry void run() {
      // ... main loop (next slide) ...
    };
  };
};
```

Jacobi: .ci file

```
while (!converged) {
  serial {
    copyToBoundaries();
    int x = thisIndex.x, y = thisIndex.y, z = thisIndex.z;
    int bdX = blockDimX, bdY = blockDimY, bdZ = blockDimZ;
    thisProxy(wrapX(x-1),y,z).updateGhosts(iter, RIGHT, bdY, bdZ, rightGhost);
    thisProxy(wrapX(x+1),y,z).updateGhosts(iter, LEFT, bdY, bdZ, leftGhost);
    thisProxy(x,wrapY(y-1),z).updateGhosts(iter, TOP, bdX, bdZ, topGhost);
    thisProxy(x,wrapY(y+1),z).updateGhosts(iter, BOTTOM, bdX, bdZ, bottomGhost);
    thisProxy(x,y,wrapZ(z-1)).updateGhosts(iter, BACK, bdX, bdY, backGhost);
    thisProxy(x,y,wrapZ(z+1)).updateGhosts(iter, FRONT, bdX, bdY, frontGhost);
    freeBoundaries();
  }
  for (remoteCount = 0; remoteCount < 6; remoteCount++)
    when updateGhosts[iter](int ref, int dir, int w, int h, double buf[w*h]) serial {
      updateBoundary(dir, w, h, buf);
    }
  serial {
    double error = computeKernel();
    int conv = error < DELTA;
    CkCallback cb(CkReductionTarget(Jacobi, checkConverged), thisProxy);
    contribute(sizeof(int), &conv, CkReduction::logical_and, cb);
  }
  when checkConverged(bool result)
    if (result) serial { mainProxy.done(iter); converged = true; }
  serial { ++iter; }
}
};
```

Jacobi: .ci file (with **asynchronous** reductions)

```
entry void run() {
  while (!converged) {
    serial {
      copyToBoundaries();
      int x = thisIndex.x, y = thisIndex.y, z = thisIndex.z;
      int bdX = blockDimX, bdY = blockDimY, bdZ = blockDimZ;
      thisProxy(wrapX(x-1),y,z).updateGhosts(iter, RIGHT, bdY, bdZ, rightGhost);
      thisProxy(wrapX(x+1),y,z).updateGhosts(iter, LEFT, bdY, bdZ, leftGhost);
      thisProxy(x,wrapY(y-1),z).updateGhosts(iter, TOP, bdX, bdZ, topGhost);
      thisProxy(x,wrapY(y+1),z).updateGhosts(iter, BOTTOM, bdX, bdZ, bottomGhost);
      thisProxy(x,y,wrapZ(z-1)).updateGhosts(iter, BACK, bdX, bdY, backGhost);
      thisProxy(x,y,wrapZ(z+1)).updateGhosts(iter, FRONT, bdX, bdY, frontGhost);
      freeBoundaries();
    }
    for (remoteCount = 0; remoteCount < 6; remoteCount++)
      when updateGhosts[iter](int ref, int dir, int w, int h, double buf[w*h]) serial {
        updateBoundary(dir, w, h, buf);
      }
    serial {
      double error = computeKernel();
      int conv = error < DELTA;
      if (iter % 5 == 1)
        contribute(sizeof(int), &conv, CkReduction::logical_and, CkCallback(CkReductionTarget(Jacobi,
                checkConverged), thisProxy));
    }
    if (++iter % 5 == 0)
      when checkConverged(bool result)
        if (result) serial { mainProxy.done(iter); converged = true; }
  }
};
```

Power of Asynchrony
Example

- Consider the following problem:
  - A large number of key-value pairs are distributed on several (hundred) processors (or chares)

Power of Asynchrony

Example

- Consider the following problem:
  - A large number of key-value pairs are distributed on several (hundred) processors (or chares)
  - Each chare needs to get some subset of these values before they can proceed to the next phase of the computation

Power of Asynchrony

Example

- Consider the following problem:
  - A large number of key-value pairs are distributed on several (hundred) processors (or chares)
  - Each chare needs to get some subset of these values before they can proceed to the next phase of the computation
  - The set of keys needed are not known in advance: they are determined based on the input data

Structured dagger version

```
entry void retrieveValues {
  for (i = 0; i < n; i++) serial {
    keys[i] = // compute i'th key;
    keyValueProxy[keys[i] / B].requestValue(keys[i], thisProxy, i);
  }
```

Structured dagger version

```
entry void retrieveValues {
  for (i = 0; i < n; i++) serial {
    keys[i] = // compute i'th key;
    keyValueProxy[keys[i] / B].requestValue(keys[i], thisProxy, i);
  }
```

```
  for (i = 0; i < n; i++)
    when response(int i, ValueType value)
      serial { values[i] = value; }
};

  // next phase of computation thats uses the keys and values.
```

Structured dagger version

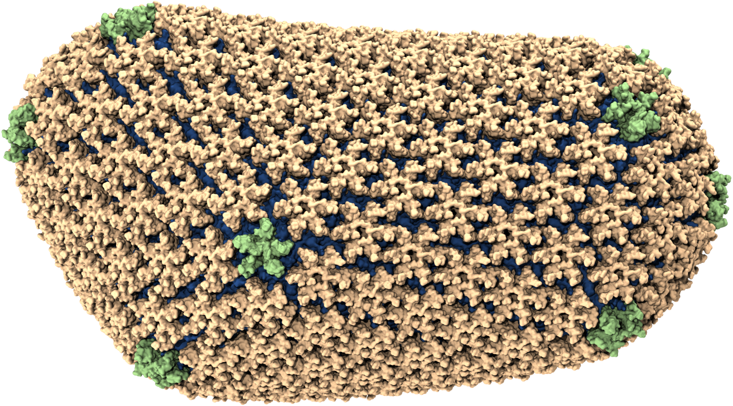
```
entry void retrieveValues {
  for (i = 0; i < n; i++) serial {
    keys[i] = // compute i'th key;
    keyValueProxy[keys[i] / B].requestValue(keys[i], thisProxy, i);
  }
```

```
  for (i = 0; i < n; i++)
    when response(int i, ValueType value)
      serial { values[i] = value; }
};

  // next phase of computation thats uses the keys and values.
```

```
KeyValueClass::requestValue(int key, CProxy_Client c, int ref) {
  ValueType v = localTable[key];
  c.response(ref, v);
}
```

Outline

NAMD



- Ground-breaking Nature article on the structure of the HIV capsid

Molecular Dynamics in NAMD

- Collection of charged atoms, with bonds
  - Newtonian mechanics
  - Relatively small of atoms (100K 10M)

Molecular Dynamics in NAMD

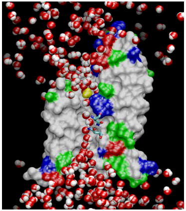
- Collection of charged atoms, with bonds
  - Newtonian mechanics
  - Relatively small of atoms (100K 10M)
- Calculate forces on each atom
  - Bonds
  - Non-bonded: electrostatic and van der Waals
    - Short-distance: every timestep
    - Long-distance: using PME (3D FFT)
    - Multiple Time Stepping : PME every 4 timesteps

Molecular Dynamics in NAMD

- Collection of charged atoms, with bonds
  - Newtonian mechanics
  - Relatively small of atoms (100K 10M)
- Calculate forces on each atom
  - Bonds
  - Non-bonded: electrostatic and van der Waals
    - Short-distance: every timestep
    - Long-distance: using PME (3D FFT)
    - Multiple Time Stepping : PME every 4 timesteps
- Calculate velocities and advance positions
- Challenge: femtosecond time-step, millions needed!

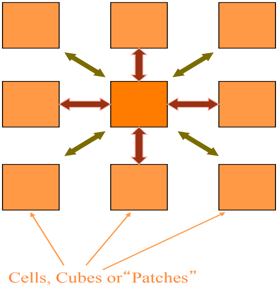
Molecular Dynamics in NAMD

- Collection of charged atoms, with bonds
  - Newtonian mechanics
  - Relatively small of atoms (100K 10M)
- Calculate forces on each atom
  - Bonds
  - Non-bonded: electrostatic and van der Waals
    - ★ Short-distance: every timestep
    - ★ Long-distance: using PME (3D FFT)
    - ★ Multiple Time Stepping : PME every 4 timesteps
- Calculate velocities and advance positions
- Challenge: femtosecond time-step, millions needed!



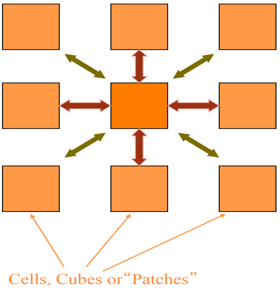
Collaboration with K. Schulten, R. Skeel, and coworkers

Spatial Decomposition Via Charm



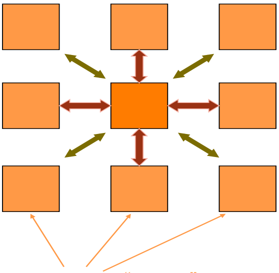
- Atoms distributed to cubes based on their location

Spatial Decomposition Via Charm



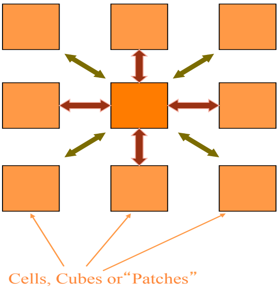
- Atoms distributed to cubes based on their location
- Size of each cube :
  - Just a bit larger than cut-off radius
  - Communicate only with neighbors
  - Work: for each pair of nbr objects
- C/C ratio: O(1)

Spatial Decomposition Via Charm



- Atoms distributed to cubes based on their location
- Size of each cube :
  - Just a bit larger than cut-off radius
  - Communicate only with neighbors
  - Work: for each pair of nbr objects
- C/C ratio: O(1)
- However:
  - Load imbalance
  - Limited parallelism

Spatial Decomposition Via Charm

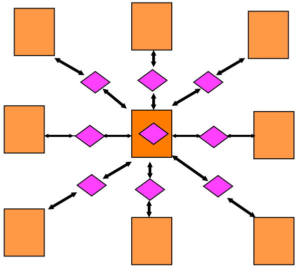


- Atoms distributed to cubes based on their location
- Size of each cube :
  - Just a bit larger than cut-off radius
  - Communicate only with neighbors
  - Work: for each pair of nbr objects
- C/C ratio: O(1)
- However:
  - Load imbalance
  - Limited parallelism

Charm++ is useful to handle this case

Object Based Parallelization for MD

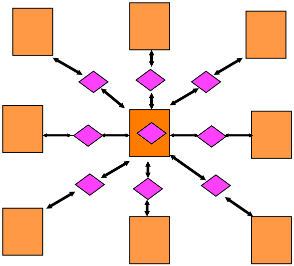
Force Decomposition + Spatial Decomposition



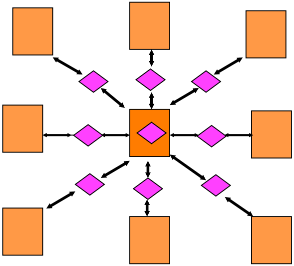
- Now, we have many objects to load balance:
  - Each diamond can be assigned to any proc.
  - Number of diamonds (3D): 14*Number of Patches

Object Based Parallelization for MD

Force Decomposition + Spatial Decomposition



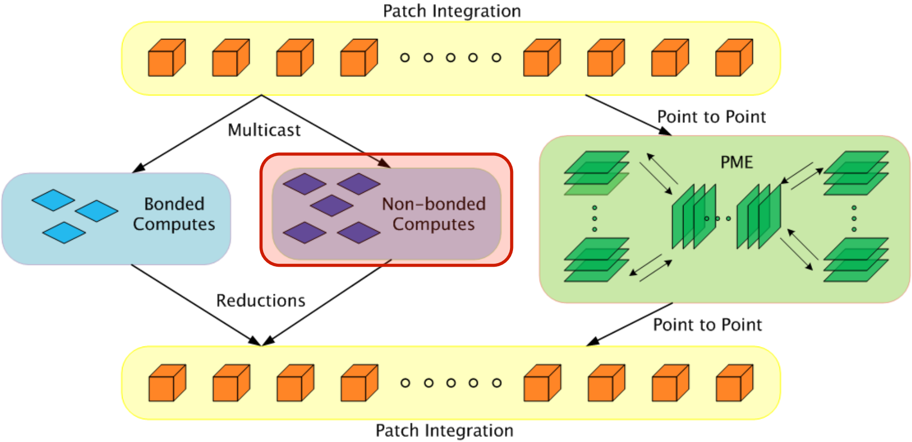
- Now, we have many objects to load balance:
  - Each diamond can be assigned to any proc.
  - Number of diamonds (3D): 14*Number of Patches
- 2-away variation:
  - Half-size cubes
  - Communicate only with neighbors
  - 5 x 5 x 5 interactions



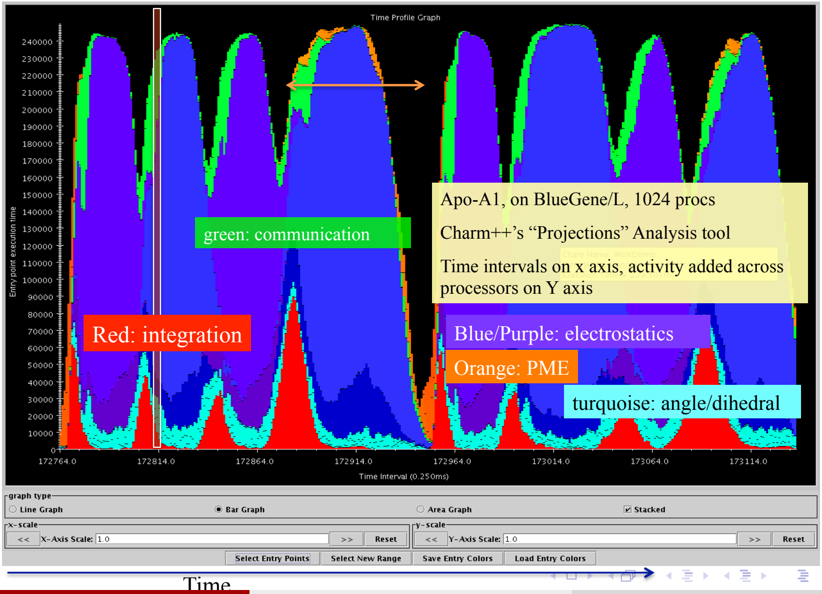
- Now, we have many objects to load balance:
  - Each diamond can be assigned to any proc.
  - Number of diamonds (3D): 14*Number of Patches
- 2-away variation:
  - Half-size cubes
  - Communicate only with neighbors
  - 5 x 5 x 5 interactions
- 3-away interactions: 7 x 7 x 7

NAMD Parallelization Using Charm++

The computation is decomposed into “natural” objects of the application, which are assigned to processors by Charm++ RTS

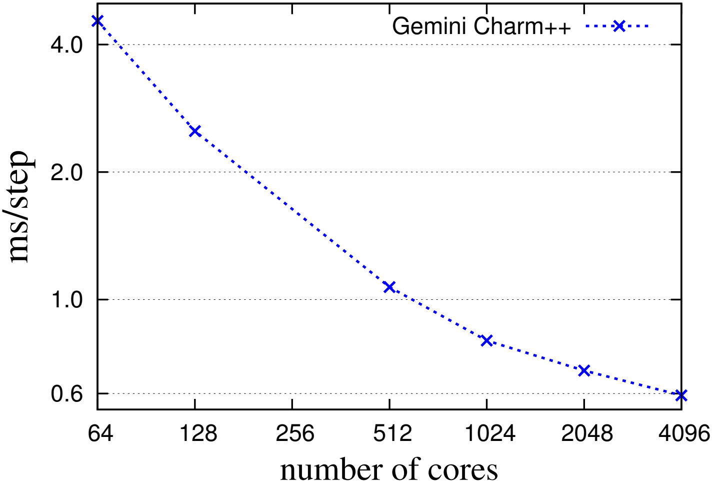


NAMD Projections



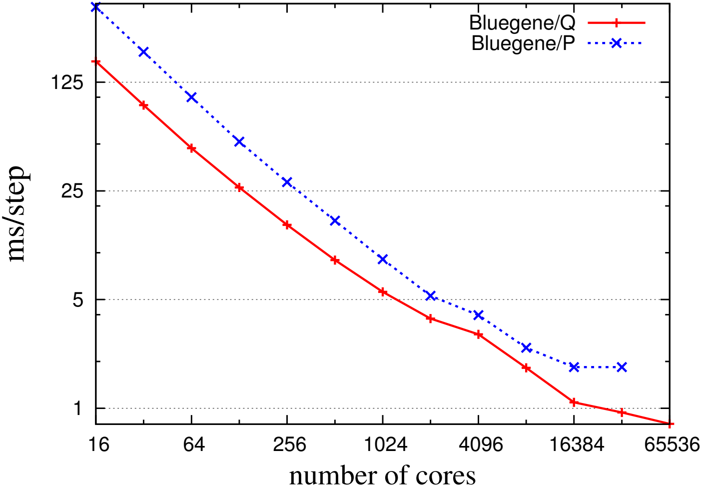
DHFR Performance on Titan

- Best performance is 590us/step



Apoa1 Performance on BG/P BG/Q

- Best performance on BG/Q is 794us/step



ChaNGa: Parallel Gravity

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

ChaNGa: Parallel Gravity

- Collaborative project (NSF)
  - with Tom Quinn, Univ. of Washington
- Evolution of Universe and Galaxy Formation
- Gravity, gas dynamics

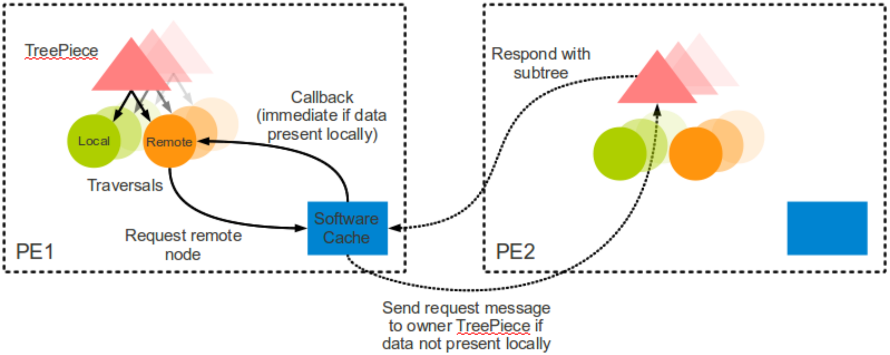
ChaNGa: Parallel Gravity

- Collaborative project (NSF)
  - with Tom Quinn, Univ. of Washington
- Evolution of Universe and Galaxy Formation
- Gravity, gas dynamics
- Barnes-Hut tree codes
  - Oct tree is natural decomposition
  - 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

ChaNGa: Parallel Gravity

- Collaborative project (NSF)
  - with Tom Quinn, Univ. of Washington
- Evolution of Universe and Galaxy Formation
- Gravity, gas dynamics
- Barnes-Hut tree codes
  - Oct tree is natural decomposition
  - 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
- With Charm++: Use Oct-Tree, and let Charm++ map subtrees to processors

ChaNGa: Control Flow



OpenAtom: MD with quantum effects

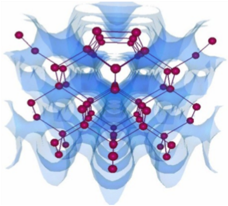
- Much more fine-grained:
  - Each electronic state is modeled with a large array

OpenAtom: MD with quantum effects

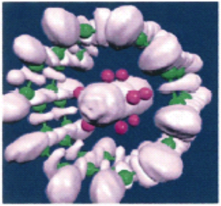
- Much more fine-grained:
  - Each electronic state is modeled with a large array
- Collaboration with:
  - G. Martyna (IBM)
  - M. Tuckerman (NYU)

OpenAtom: MD with quantum effects

- Much more fine-grained:
  - Each electronic state is modeled with a large array
- Collaboration with:
  - G. Martyna (IBM)
  - M. Tuckerman (NYU)
- Using Charm++ virtualization, we can efficiently scale small (32 molecule) systems to thousands of processors

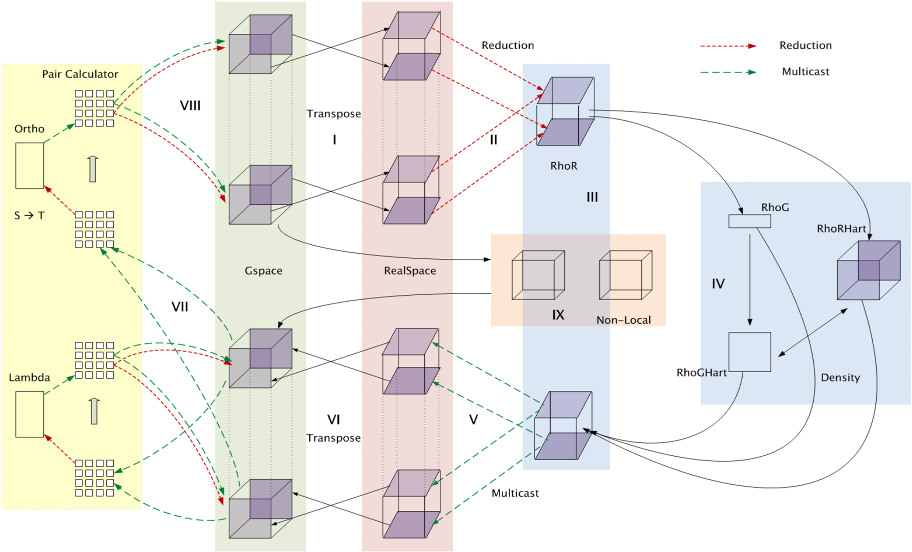


Semiconductor Surfaces



Nanowires

OpenAtom: Decomposition and Computation Flow



Outline

Performance Analysis Using Projections

- Instrumentation and measurement
  - Link program with `-tracemode projections` or `summary`
  - Trace data is generated automatically during run
  - User events can be easily inserted as needed
- Projections: visualization and analysis
  - Scalable tool to analyze up to 300,000 log files
  - A rich set of tool features : time profile, time lines, usage profile, histogram, extrema tool
  - Detect performance problems: load imbalance, grain size, communication bottleneck, etc

Using Projections

- Tools of aggregated performance viewing
  - Time profile
  - Histogram
  - Communication over time
- Tools of processor level granularity
  - Overview
  - Timeline
- Tools of derived/processed data
  - Extrema analysis : identifies outliers
  - Noise miner : highlights probable interference

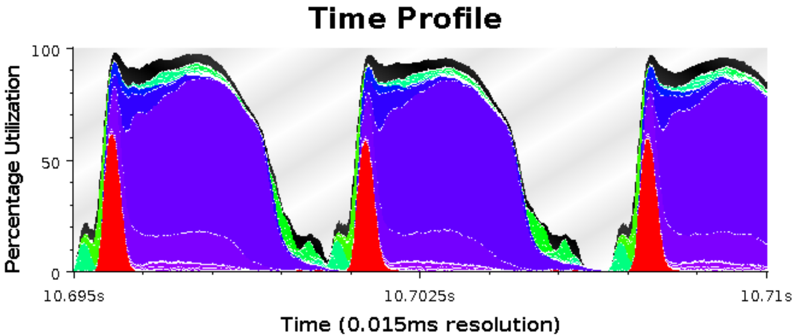
Problem Identification

- Load imbalance
  - Time profile : lower CPU usage
  - Extrema analysis tool:
    - Least idle processors
  - Load the over-loaded processors in Timeline
  - Histogram : grain size issues

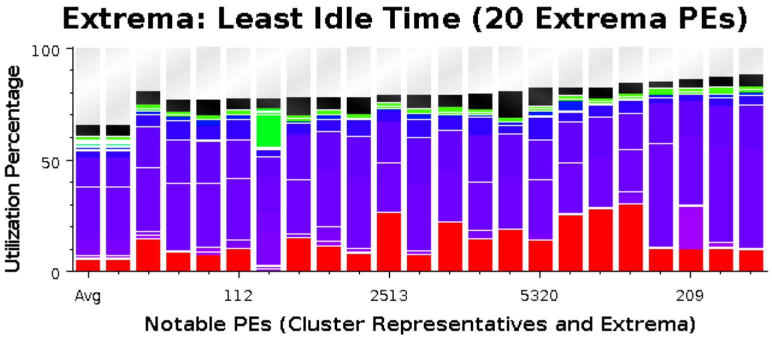
Using Projections

- Example Demonstration
  - Trying to identify the next performance obstacle for NAMD
    - Running on 8192 processors, with 1 million atom simulation
    - Jaguar Cray XK6
    - Test scenario: with PME every step

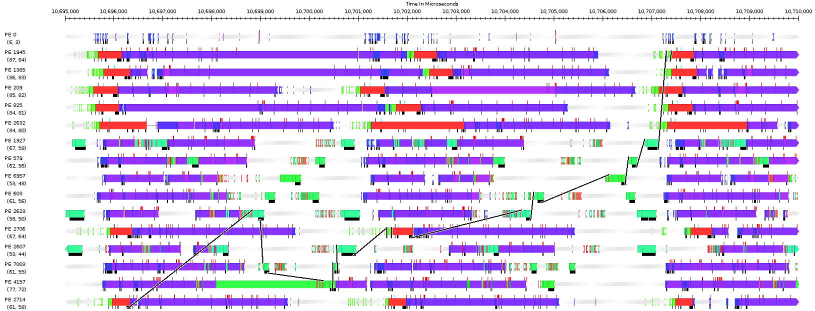
Time Profile



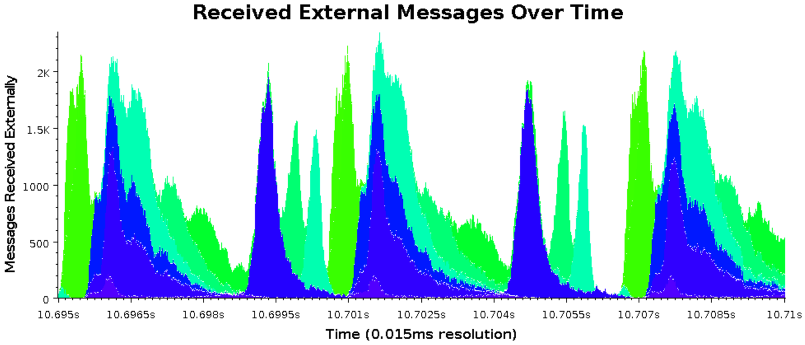
Extrema Tool for Least Idle Processors



Time Lines with Message Back Tracing



Communication over Time for all Processors



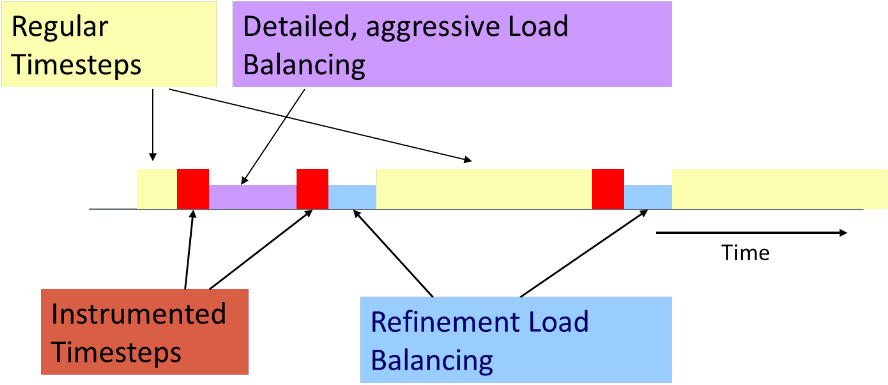
Outline

Dynamic Load Balancing

Automatic Dynamic Load Balancing

- Measurement based load balancers
  - Principle of persistence: In many CSE applications, computational loads and communication patterns tend to persist, even in dynamic computations
  - Therefore, recent past is a good predictor of near future
  - Charm++ provides a suite of load-balancers
  - Periodic measurement and migration of objects
- Seed balancers (for task-parallelism)
  - Useful for divide-and-conquer and state-space-search applications
  - Seeds for charm++ objects moved around until they take root

Typical Load Balancing Steps



Code to Use Load Balancing

- Write PUP method to serialize the state of a chare
- Insert `if (myLBStep) AtSync();` call at natural barrier
- Implement `ResumeFromSync()` to resume execution
  - Typical ResumeFromSync contribute to a reduction

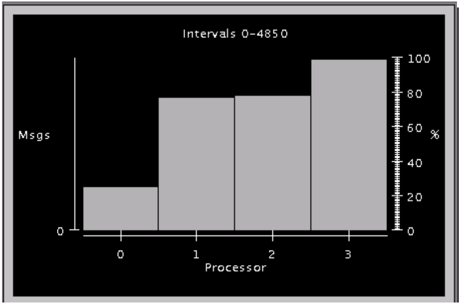
Using the Load Balancer

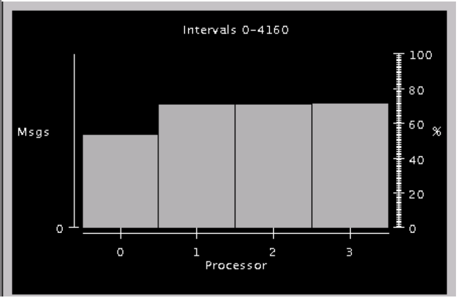
- link a LB module
  - `-module <strategy>`
  - RefineLB, NeighborLB, GreedyCommLB, others
  - EveryLB will include all load balancing strategies
- compile time option (specify default balancer)
  - `-balancer RefineLB`
  - runtime option
  - `+balancer RefineLB`

Example: Stencil

```
while (!converged) {
  serial {
    int x = thisIndex.x, y = thisIndex.y, z = thisIndex.z;
    copyToBoundaries();
    thisProxy(wrapX(x−1),y,z).updateGhosts(i, RIGHT, dimY, dimZ, right);
    /* ...similar calls to send the 6 boundaries... */
    thisProxy(x,y,wrapZ(z+1)).updateGhosts(i, FRONT, dimX, dimY, front);
  }
  for (remoteCount = 0; remoteCount < 6; remoteCount++) {
    when updateGhosts[i](int i, int d, int w, int h, double b[w*h])
    serial { updateBoundary(d, w, h, b); }
  }
  serial {
    int c = computeKernel() < DELTA;
    CkCallback cb(CkReductionTarget(Jacobi, checkConverged), thisProxy);
    if (i%5 == 1) contribute(sizeof(int), \&c, CkReduction::logical_and, cb);
  }
  if (i % lbPeriod == 0) { serial { AtSync(); } when ResumeFromSync() {} }
  if (++i % 5 == 0) {
    when checkConverged(bool result) serial {
      if (result) { mainProxy.done(); converged = true; }
    }
  }
}
```

Performance





How to Diagnose Load Imbalance

- Often hidden in statements such as:
  - Very high synchronization overhead
    - Most processors are waiting at a reduction
- Count total amount of computation (ops/flops) per processor
  - In each phase!
  - Because the balance may change from phase to phase

Golden Rule of Load Balancing

Fallacy: objective of load balancing is to minimize variance in load across processors

Example:

- 50,000 tasks of equal size, 500 processors:
  - A: All processors get 99, except last 5 gets $100 + 99 = 199$
  - OR, B: All processors have 101, except last 5 get 1

Identical variance, but situation A is much worse!

Golden Rule: It is ok if a few processors idle, but avoid having processors that are overloaded with work

Finish time $= max_i$(Time on processor i)

excepting data dependence and communication overhead issues

The speed of any group is the speed of slowest member of that group.

Dynamic Load Balancing Scenarios

- Examples representing typical classes of situations
  - Particles distributed over simulation space
    - Dynamic: because Particles move.
    - Cases:
      Highly non-uniform distribution (cosmology)
      Relatively Uniform distribution
- Structured grids, with dynamic refinements/coarsening
- Unstructured grids with dynamic refinements/coarsening

Load Balancing Strategies

- Classified by when it is done:
  - Initially
  - Dynamic: Periodically
  - Dynamic: Continuously
- Classified by whether decisions are taken with global information
  - Fully centralized
    - Quite good a choice when load balancing period is high
  - Fully distributed
    - Each processor knows only about a constant number of neighbors
    - Extreme case: totally local decision (send work to a random destination processor, with some probability).
  - Use *aggregated* global information, and *detailed* neighborhood info.

Example Case: Particles

Orthogonal Recursive Bisection (ORB)

- At each stage: divide Particles equally
- Processor dont need to be a power of 2:
  - Divide in proportion
    - 2:3 with 5 processors
- How to choose the dimension along which to cut?
  - Choose the longest one
- How to draw the line?
  - All data on one processor? Sort along each dimension
  - Otherwise: run a distributed histogramming algorithm to find the line, recursively
- Find the entire tree, and then do all data movement at once
  - Or do it in two-three steps.
  - But no reason to redistribute particles after drawing each line.

Periodic Load Balancing

Centralized strategies:

- Charm RTS collects data (on one processor) about:
  - Computational Load and Communication for each pair
- Partition the graph of objects across processors
  - Take communication into account
    - Pt-to-pt, as well as multicast over a subset
    - As you map an object, add to the load on both sending and receiving processor
  - Multicasts to multiple co-located objects are effectively the cost of a single send

Object Partitioning Strategies

- You can use graph partitioners like METIS, K-R
  - BUT: graphs are smaller, and optimization criteria are different
- Greedy strategies:
  - If communication costs are low: use a simple greedy strategy
    - ★ Sort objects by decreasing load
    - ★ Maintain processors in a heap (by assigned load)
    - ★ In each step:
    - ★ assign the heaviest remaining object to the least loaded processor
  - With small-to-moderate communication cost:
    - ★ Same strategy, but add communication costs as you add an object to a processor
  - Always add a refinement step at the end:
    - ★ Swap work from heaviest loaded processor to "some other processor"
    - ★ Repeat a few times or until no improvement

Object Partitioning Strategies 2

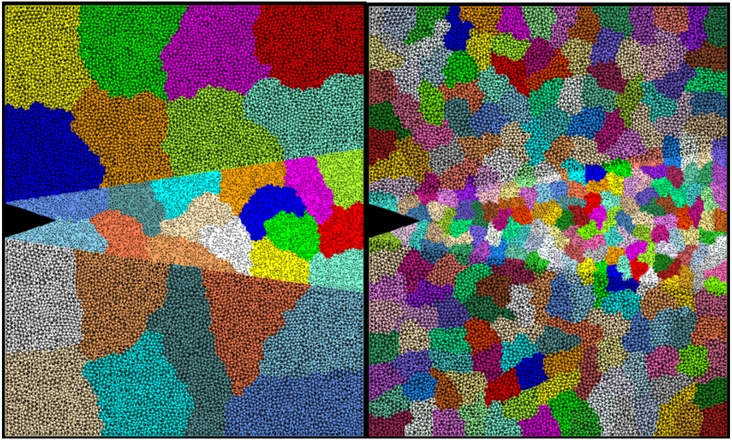
When communication cost is significant:

- Still use greedy strategy, but:
  - At each assignment step, choose between assigning O to least loaded processor and the processor that already has objects that communicate most with O.
    - ★ Based on the degree of difference in the two metrics
    - ★ Two-stage assignments:
      In early stages, consider communication costs as long as the processors are in the same (broad) load class,
      In later stages, decide based on load

Branch-and-bound

- Searches for optimal, but can be stopped after a fixed time

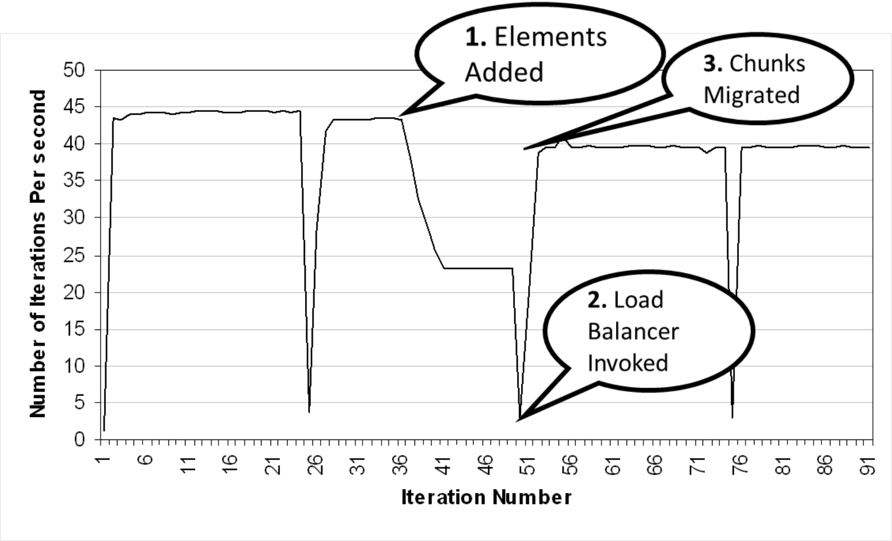
Crack Propagation



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

As computation progresses, crack propagates, and new elements are added, leading to more complex computations in some chunks.

Load Balancing Crack Propagation



Distributed Load balancing

- Centralized strategies
  - Still ok for 3000 processors for NAMD
- Distributed balancing is needed when:
  - Number of processors is large and/or
  - load variation is rapid
- Large machines:
  - Need to handle locality of communication
    - Topology sensitive placement
  - Need to work with scant global information
    - Approximate or aggregated global information (average/max load)
    - Incomplete global info (only neighborhood)
    - Work diffusion strategies (1980s work by Kale and others!)
  - Achieving global effects by local action

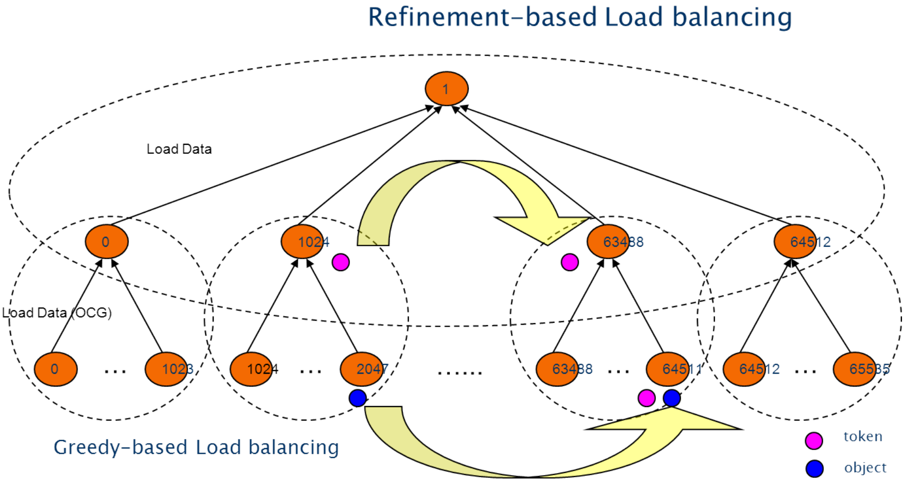
Load Balancing on Large Machines

- Centralized load balancing strategies dont scale on extremely large machines
- Limitations of centralized strategies:
    - Central node: memory/communication bottleneck
    - Decision-making algorithms tend to be very slow
- Limitations of distributed strategies:
    - Difficult to achieve well-informed load balancing decisions

Hierarchical Load Balancers

- Partition processor allocation into processor groups
- Apply different strategies at each level
- Scalable to a large number of processors

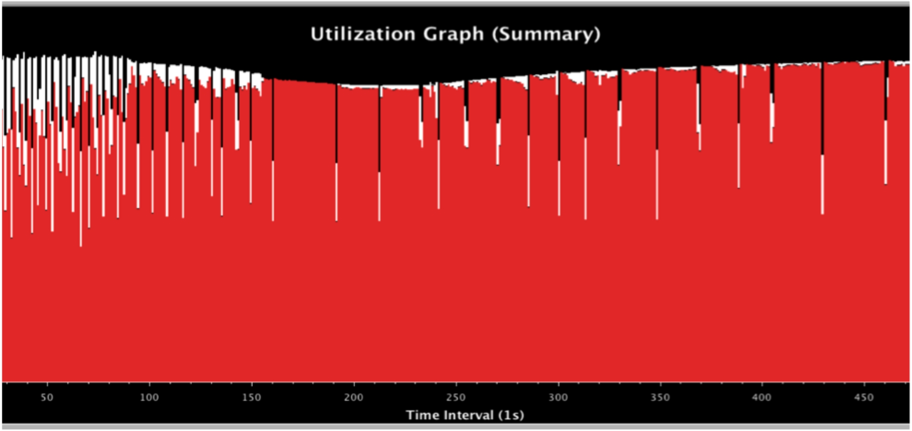
Our Hybrid Scheme



MetaBalancer - When and how to load balance?

- Difficult to find the optimum load balancing period
  - Depends on the application characteristics
  - Depends on the machine the application is run on
- Monitors the application continuously and predicts behavior.
- Decides when to invoke which load balancer.
- Command line argument - `+MetaLB`

Metabalancer Utilization Graph for Fractography



Outline

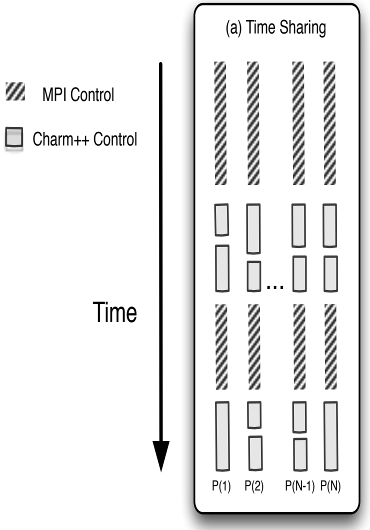
Adaptive MPI

- MPI implemented on top of Charm++
- Each MPI process implemented as a user-level thread embedded in a chare
- Overdecompose to obtain communication-computation overlap between threads
- Supports migration, load balancing, fault tolerance and other Charm++ functionality
- Use cases - Rocstar, BRAMS, NPB, Lulesh etc
- Build with AMPI as target and compile using ampi* compilers
  ./build AMPI net-linux-x86_64 –with-production –enable-tracing -j8
  ampiCC myAMPIpgm.C -o myAMPIpgm

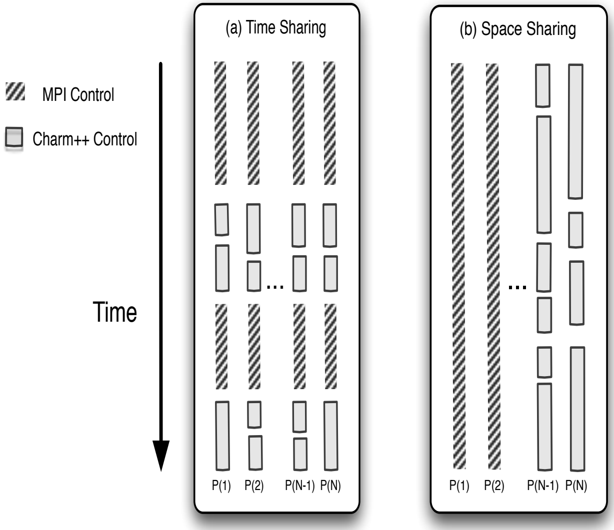
Charm++-MPI Interoperability

- Any library written in Charm++ can be called from MPI
- Charm++ resides in the same memory space as the MPI program
- Control transfer between MPI and Charm++ analogous to the control transfer between a program and an external library being used by the program
- Currently requires mpi-based build of Charm++

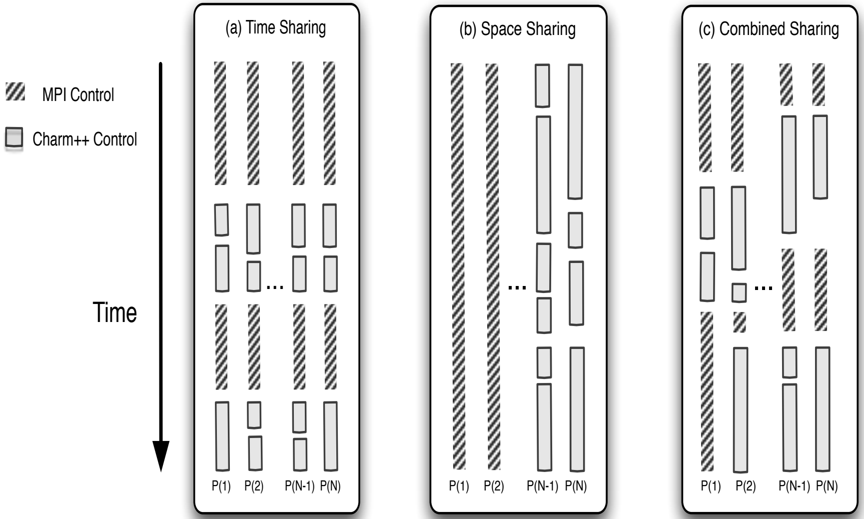
Interoperability Modes



Interoperability Modes



Interoperability Modes



Example Code Flow

```
MPI_Init(argc,argv); //initialize MPI
//Do MPI related work here
```

```
//create comm to be used by Charm++
MPI_Comm_split(MPI_COMM_WORLD, myRank % 2, myRank, newComm);
CharmLibInit(newComm,.) //initialize Charm++ over my communicator
```

```
if(myRank % 2)
  StartHello(); //invoke Charm++ library on one set
else
  //do MPI work on other set
```

```
kNeighbor(); //invoke Charm++ library on both sets individually
CharmLibExit(); //destroy Charm++
```

Enabling Interoperability

- Add interface functions that can be called from MPI, and triggers Charm++ RTS-

```
void StartHello(int elems)
  if(CkMyPe() == 0) {
    CProxy_MainHello mainhello =
    CProxy_MainHello::ckNew(elems);
  }
  StartCharmScheduler();
}
```

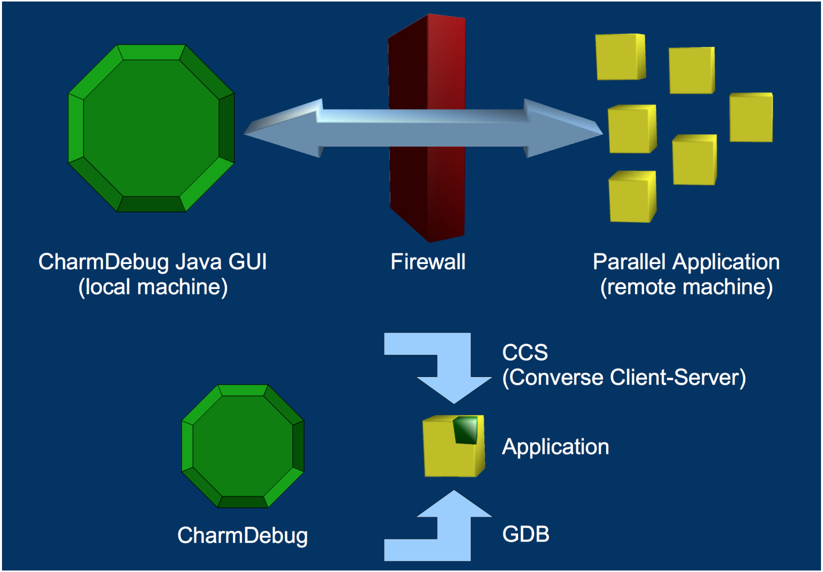
- Use CkExit to return the control back to MPI
- Include *mpi-interoperate.h* in MPI and Charm++ code

Outline

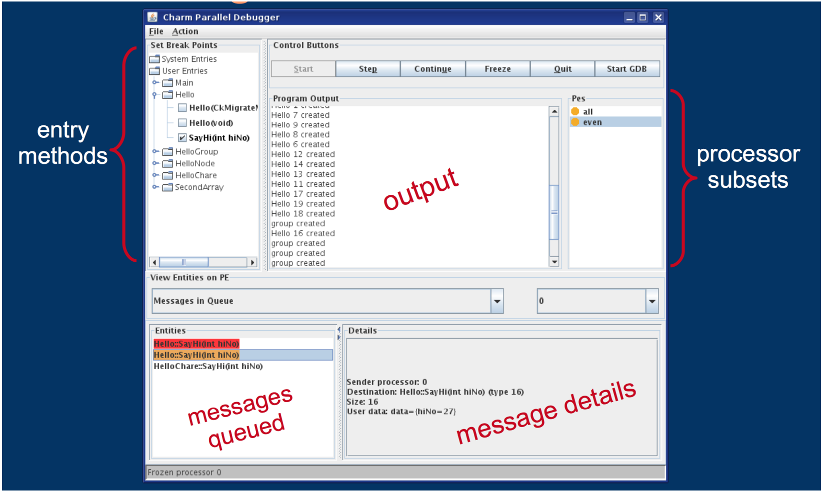
Debugging Parallel Applications

- It can be very difficult
- The typical "printf" strategy may be insufficient
- Using `gdb`
  - Very easy with Charm++!
  - Just run the application with the `++debug` command line parameter and a `gdb` window for each PE will open through X (and can be forwarded)
    - Not very scalable
- We have developed a scalable tool for debugging Charm++ applications
  - It's interactive
  - Allows you to change message order to find bugs!
  - "What-if" scenarios can be explored using provisional message delivery
  - Memory can be tracked to find memory leaks

Overview of CharmDebug



CharmDebug


entry methods
Charm Parallel Debugger
File Action
Set Break Points
System Entries
User Entries
Main
Hello
Hello(CkMigrateM
Hello(void)
SayHi(int hiNo)
HelloGroup
HelloNode
HelloChare
SecondArray
Control Buttons
Start
Step
Continue
Freeze
Quit
Start GDB
Program Output
Hello 7 created
Hello 9 created
Hello 8 created
Hello 6 created
Hello 12 created
Hello 14 created
Hello 13 created
Hello 11 created
Hello 17 created
Hello 19 created
Hello 18 created
group created
Hello 16 created
group created
group created
group created
output
Pes
all
even
processor subsets
View Entities on PE
Messages in Queue
0
Entities
Hello::SayHi(int hiNo)
Hello::SayHi(int hiNo)
HelloChare::SayHi(int hiNo)
messages queued
Details
Sender processor: 0
Destination: Hello::SayHi(int hiNo) (type 16)
Size: 16
User data: data={hiNo=27}
message details
Frozen processor 0

Getting CharmDebug

- It is part of Charm++
- For the basic feature set, nothing special needs to be done
- Precompiled for java 6
  - Use `ant` to recompile
- Help
  - `charm@cs.illinois.edu` (preferred)
  - `ppl@cs.illinois.edu`

Compiling Your Applications for use with CharmDebug

- Charm++
  - Use `-g`
  - No `-O3` or `--with-production`
- Application
  - Just compile with `-g`
  - OR
  - Compile with `-debug`
    - Adds `-g -O0`, `--memory charmdebug`, Python modules

Launching in Debug Mode

- Attach to running application in net- build
  - Uses CCS to receive application output
- Attach to running application in other builds
  - Read the output file of the application
- Start a new application in net- build
  - Can use tunnels
- Options available also in command line
  - Use `charmdebug help` to see them

Outline

Overview of Performance Enhancement Features

Quiesence Detection

- What if determining global termination of an application is difficult?
- Mechanism to detect completion - Quiesence!
- From any chare, invoke
  CkStartQD(CkCallback(CkIndex_Main::finished(), mainProxy));
- Runs in background, waits for all outstanding messages to be consumed.
- Invokes the callback when quiesence is detected.

Shared Memory Optimizations

- Objects' memory buffers disjoint
- Communication will leverage refcounted message pointers to avoid copying
- Avoids packing/unpacking within node
- Single copy of node level read only structures
- Dedicated thread for intra-node communication

Controlling Placement

- In some applications, load patterns dont change much as computation progresses
  - You, the programmer, may want to control which chare lives on which processors
  - This is also true when load may evolve over time, but you want to control initial placement of chares
- The feature in Charm++ for this purpose is called Map Objects
  - Sec. 13.2.2 of the Charm++ manual

Messages

- Avoids extra copy
- Can be custom packed
- Reusable
- Useful for transfer of complex data structures
- It provides explicit control for the application over allocation, reuse, and scope
- Encapsulates variable size quantities
- Execution order of messages in the queue can be prioritized

Groups

- Like a chare-array with one chare per PE
- Encapsulate processor local data
- May access the local member as a regular C++ object
- In .ci file,

```
group ExampleGroup {
   // Interface specifications as for normal chares
   // For instance, the constructor ...
   entry ExampleGroup(parameters1);
   // ... and an entry method
   entry void someEntryMethod(parameters2);
};
```

- No difference in .h and .C file definitions

Node Groups

- A chare-array with one chare per node
  - In non-smp node groups and node groups are same
- No difference in .h and .C
- Creation and usage same as others
- An entry method on a node-group member may be executed on any PE of the node
- Concurrent execution of two entry methods of a node-group member may happen
  - Use `[exclusive]` for entry methods which are unsuitable for reentrance safety

Customizing Entry Method Attributes

- `threaded` executed using separate thread
  - each thread has a stack, and may be suspended, for sync methods or futures
  - to set stacks size use +stacksize $<$ size in bytes $>$
- `sync` - returns a value
- `inline` entry method invoked immediately if destination chare on same PE
  - blocking call
- `reductiontarget` target of an array reduction
  - Takes parameter marshaled arguments
- `notrace` not traced for projections

Customizing Entry Methods

- `expedited` entry method skips the priority-based message queue in Charm++ runtime (for groups)
- `immediate` - skips the message scheduling queue (for any chare array)
- `nokeep` message belongs to Charm
- `exclusive` mutual exclusion on execution of entry methods on node-groups
- `python` can be called from python scripts

Sections

- It is often convenient to define subcollections of elements within a chare array
  - Example: rows or columns of a 2D chare array
  - One may wish to perform collective operations on the subcollection (e.g. broadcast, reduction)
- Sections are the standard subcollection construct in Charm++

```
CProxySection_Hello proxy =
  CProxySection_Hello::ckNew(helloArrayID, 0, 9, 1, 0, 19, 2, 0, 29, 2);
```

`sync` methods

- Synchronous as opposed to asynchronous
- They return a value - always a `message` type
- Other than that, just like any other entry method:

In the interface file:

```
entry [sync] MsgData * f(double A[2*m], int m );
```

In the C++ file:

```
MsgData *f(double X[], int size) {
  ...
  m = new MsgData(..);
  ...
  return m;
}
```

Threaded methods

- Any method that calls a sync method must be able to suspend:
  - Needs to be declared as a `threaded` method
  - A threaded method of a chare C
    - Can suspend, without blocking the processor
    - Other chares can then be executed
    - Even other methods of chare C can be executed
- Low level thread operations for advanced users:
  - `CthThread CthSelf()`
  - `CthAwaken(CthThread t)`
  - `CthYield()`
  - `CthSuspend()`

Customized Load Balancers

- Statistics collected by Charm

```
struct LDStats { // load balancing database
  ProcStats *procs; //statistics of PEs
  int count;

  int n_objs;
  int n_migrateobjs;
  LDObjData* objData; //info regarding chares

  int n_comm;
  LDCommData* commData; //communication information

  int *from_proc, *to_proc; //residence of
  chares
}
```

- Use LDStats, ProcArray and ObjGraph for processor load and communication statistics
- *work* is the function invoked by Charm RTS to perform load balancing

Conclusion

- Charm++ is a production-ready parallel programming system
- Program mostly in C++
- Very powerful runtime system
  - Dynamic load balancing
  - Automatic overlap of computation and communication
  - Fault tolerance built in
- Topics we did not cover:
  - Many different types of load balancers
  - Threaded methods in detail
  - Futures
  - Accelerator support
  - Topology aware communication strategies
- More information on `http://charm.cs.illinois.edu/`