Programming with Parallel Objects

Session 1: Fundamentals of Parallel Objects Programming

Esteban Meneses

Advanced Computing Laboratory
Costa Rica National High Technology Center

School of Computing
Costa Rica Institute of Technology

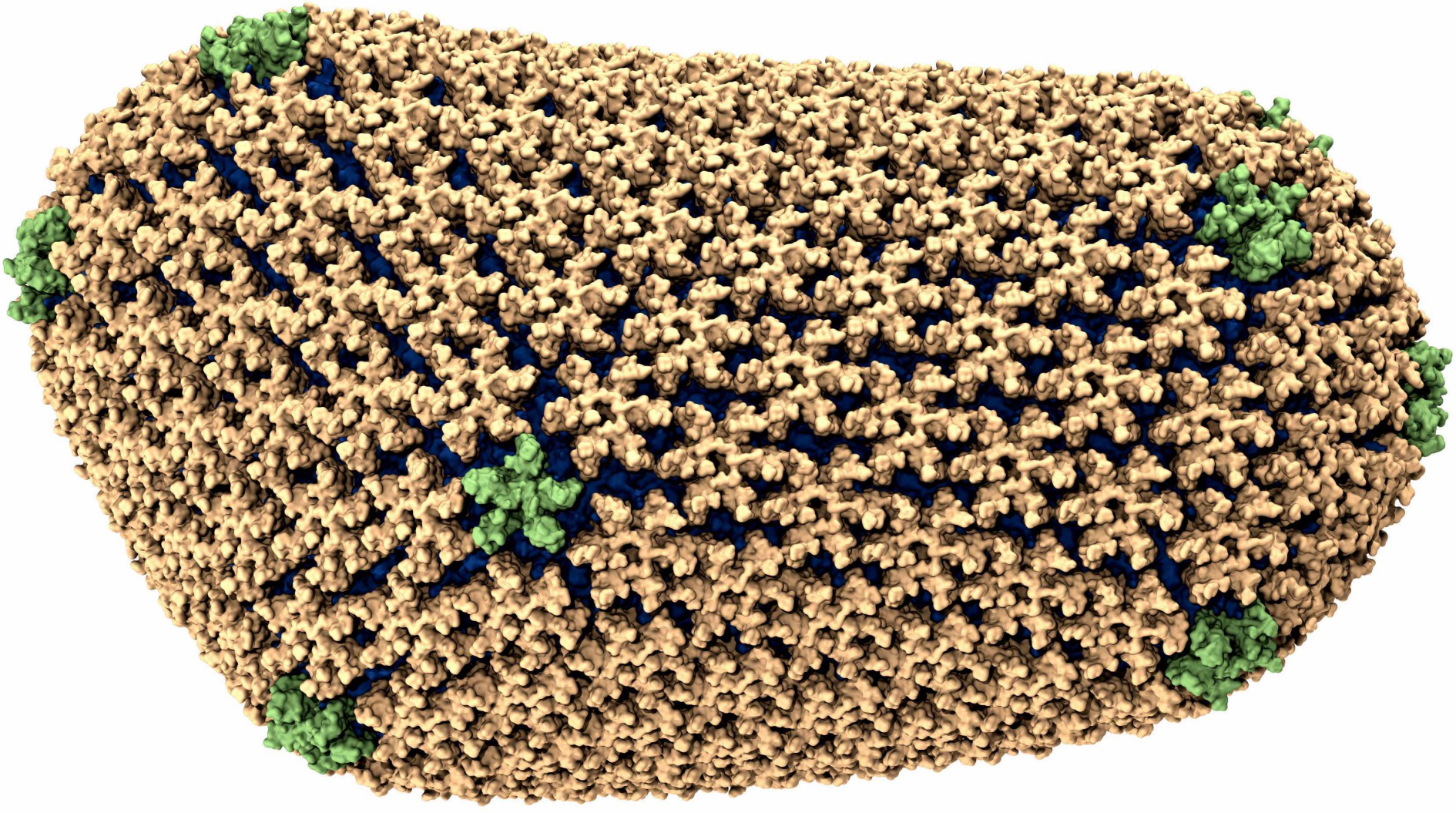
2017

ACKNOWLEDGEMENTS

Most of the content of this tutorial was provided courtesy of Prof. Laxmikant V. Kalé from the Parallel Programming Laboratory (PPL) of the University of Illinois at Urbana-Champaign. The PPL originally developed the ideas behind parallel object programming and has maintained the Charm++ code base for more than 20 years.

ADMINISTRATIVIA

- Official Charm++ website:
  http://www.charmplusplus.org/
- Slides and code for this tutorial:
  https://github.com/emenesesrojas/parallel_objects.git
- Questions on parallel objects programming:
  https://ecar2017.slack.com
- Advanced questions on Charm++:
  ppl@cs.illinois.edu
- Official instructor's email address:
  emeneses@cenat.ac.cr



Structure of HIV capsid

Theoretical and Computational Biophysics Group
University of Illinois at Urbana-Champaign

Outline

Challenges in High Performance Computing

Introduction
- Object Design
- Execution Model

Hello World

Benefits of Charm++

Charm++ Programming Language
- Charm++ Basics
- Object Collections

CeNAT

Harnessing Parallelism

Trends in system architecture

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

Harnessing Parallelism

High concurrency

- Compute resources are not faster cores, but **more cores**
- Unprecedented levels of available concurrency:

Rank*	Machine	Cores
1	NRCPC Sunway TaihuLight	10,649,600
2	NUDT Tianhe-2	3,120,000
5	IBM BG/Q Sequoia	1,572,864
8	Fujitsu K computer	705,024
6	Cray XC40 Cori	622,336

* Top 500 list, June 2017

- Mid-size clusters will be ubiquitous
- 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

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
- Consequences:
  - Must support automated resource management
  - Must support interoperability and parallel composition

Harnessing Parallelism

MPI programming model

- 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.)

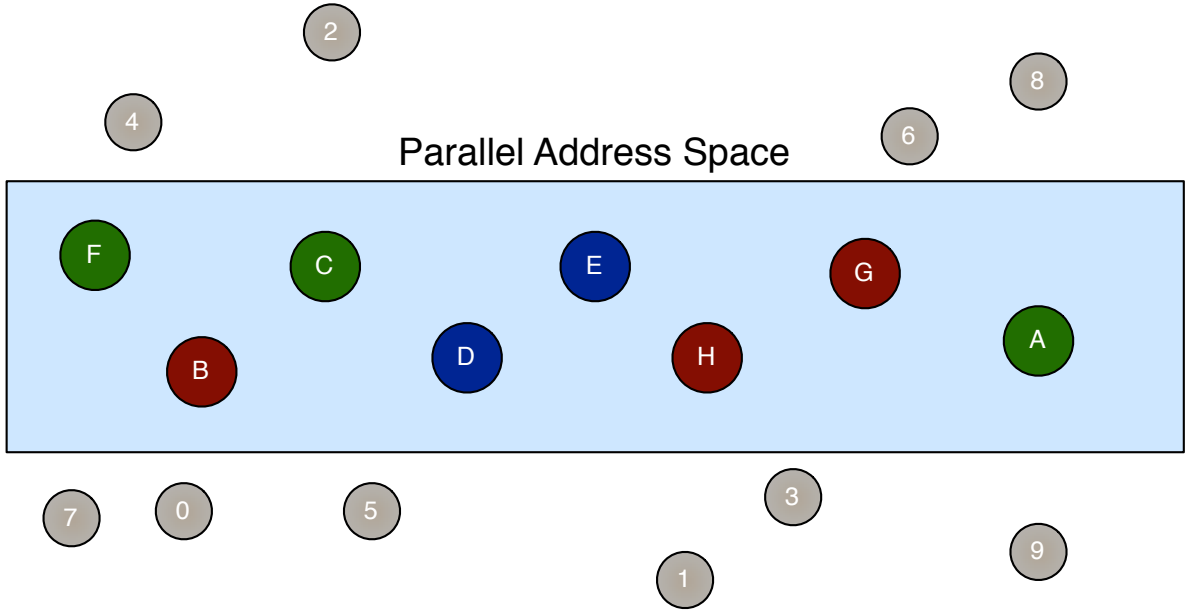
Stuff you Already Know

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
- It's not about language syntax, it's about program structure

Globally-visible Objects

Called chares (from "chores")

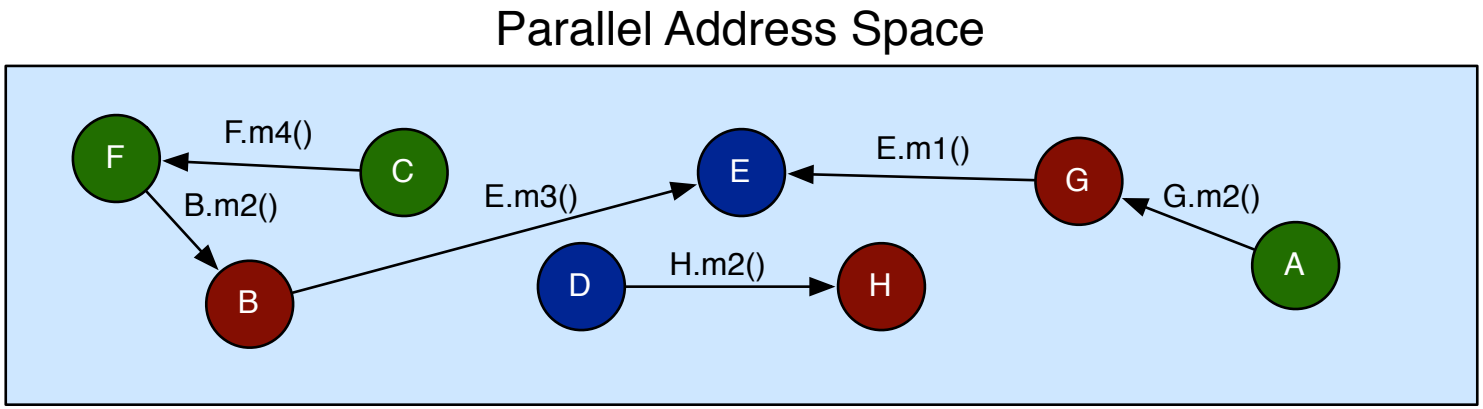


- Certain "special" object *instances* are:
  - first-class citizens in the parallel address space
  - with unique location-independent names
- Under the hood, the runtime handles locality and provides the mechanisms to promote objects to the parallel space

CeNAT

Globally-visible Methods

Entry methods

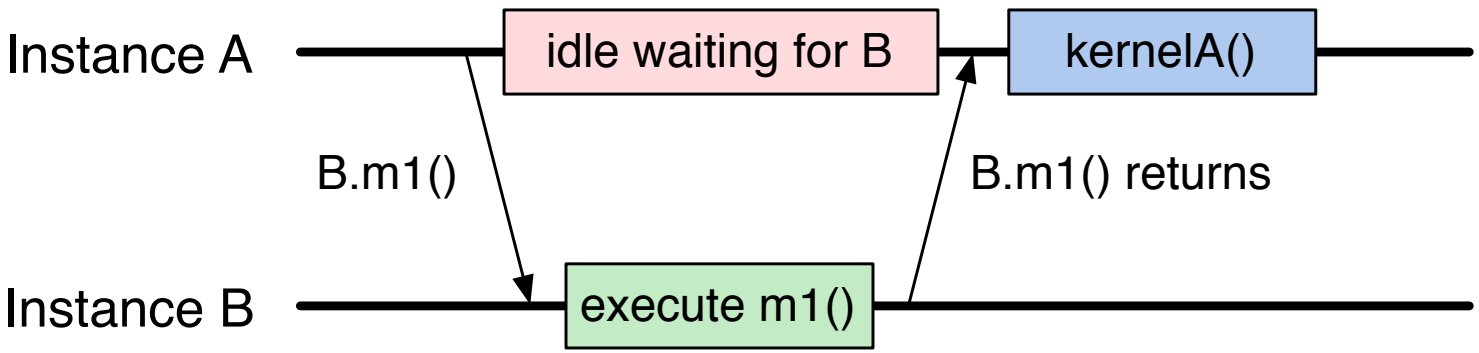


- How can objects communicate across address spaces?
  - Just like a sequential object-oriented language, an object's reference is used to invoke a method
  - In the parallel space, this is a handle that is location transparent
  - A method invocation becomes an act of communication

CeNAT

Method-driven Asynchronous Communication

Message-driven execution

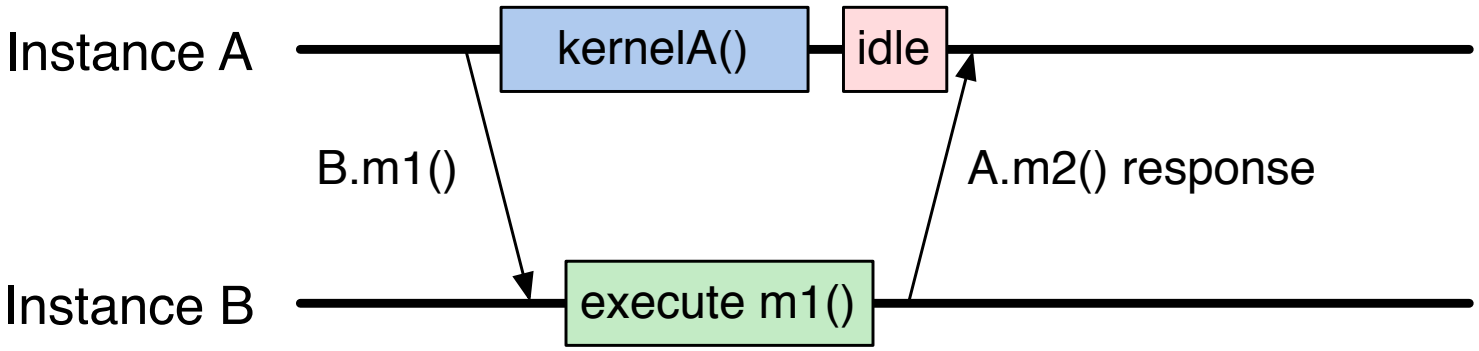


- What happens if an object waits for a return value from a method invocation?
  - Performance
  - Latency
  - Reasoning about correctness

CeNAT

Design Principle

Do not wait for remote completion

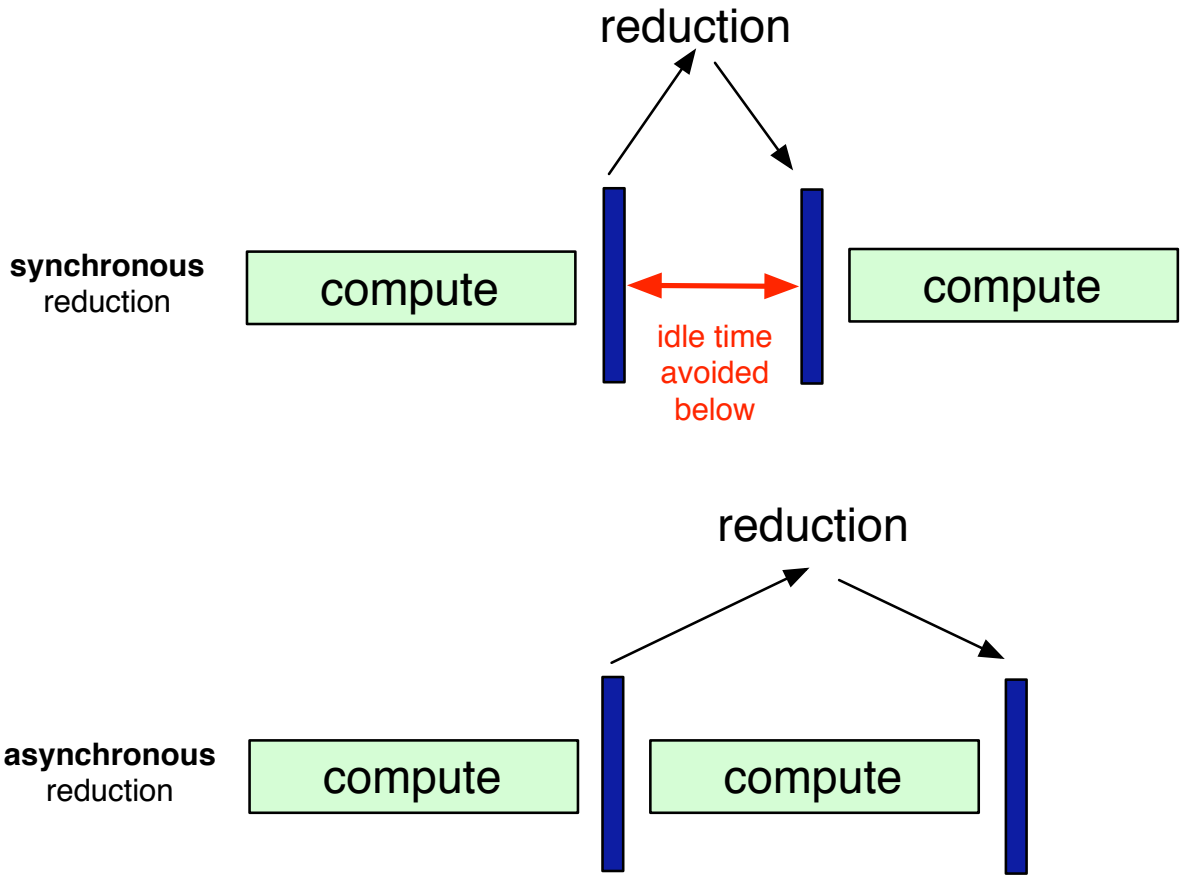


- Hence, method invocations should be asynchronous
  - No return values
- Computations are driven by the incoming data
  - Initiated by the sender or method caller

CeNAT

Example

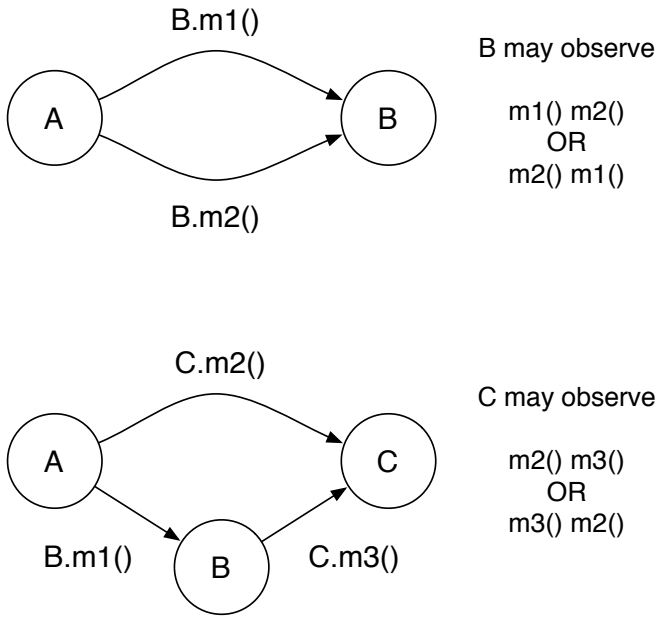
Jacobi reduction



Methods

Natural units of sequential computation

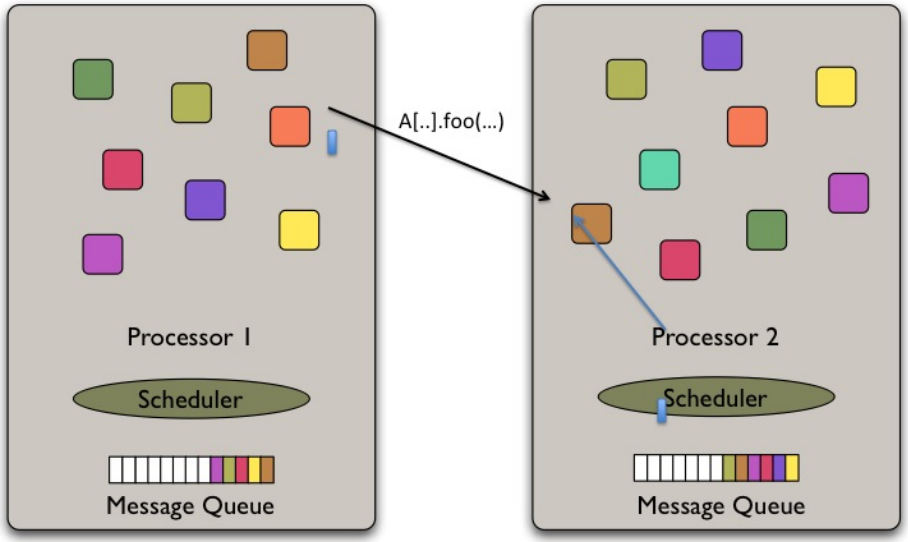
- Methods still have the same sequential semantics
  - Atomicity: methods do not execute in parallel
- Methods cannot be interrupted or preempted
- Methods interact and update state of an object in the same way
- Method sequencing is what changes from sequential computation



The Execution Model

Messages and schedulers

- Several objects live on a single *processing entity (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
  - 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

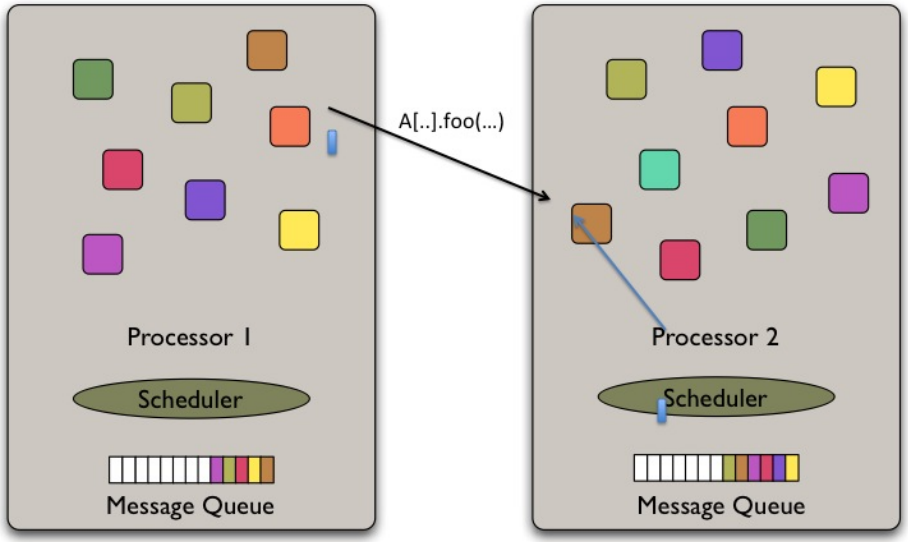


CeNAT

Message-driven Execution

Objects are reactive entities

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



Example

Hello World

- `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"
```

Example

Hello World with chares

`hello.ci` file

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

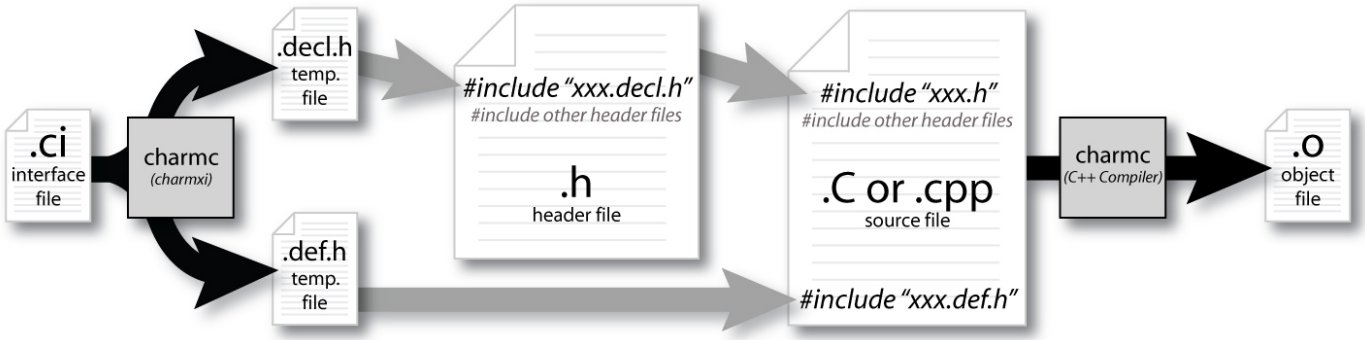
`hello.cpp` file

```
#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"
```


CeNAT

Compiling a Charm++ Program

Charm++ and C++ compilers



CeNAT

Compilation Example

Hello World

- 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

Building Charm++

1. Download the latest stable version of the code:

   `http://charmplusplus.org/download/`

2. If you are feeling adventurous, get the latest version:

   `git clone http://charm.cs.uiuc.edu/gerrit/charm`

3. Build an appropriate version of the code for you architecture:

   `./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.`

4. Get familiar with Charm++:

   `http://charmplusplus.org/tutorial/`

5. Ask questions:

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

CeNAT

Exercise 1.1

Building Charm++

Download Charm++ code base and build it for your target machine (laptop, desktop, server, supercomputer)

Examples:

- Linux cluster:

```
./build charm++ netlrts-linux-x86_64 gcc -j8 --with-production
```

- Linux cluster on MPI:

```
./build charm++ mpi-linux-x86_64 -j8 --with-production
```

- Linux laptop:

```
./build charm++ multicore-linux-x86_64 -j8 --with-production
```

- Mac laptop:

```
./build charm++ multicore-darwin-x86_64 -j8 --with-production
```

- Windows laptop with VC++:

```
./build charm++ multicore-win-x86_64 --with-production
```

Exercise 1.2

Running Charm++ Hello World

Get into Charm++'s source code and look for the Hello World example:

`charm-6.8.0/examples/charm++/hello`

Run the example

Impact on Communication

Alleviating one performance bottleneck

- Current use of communication network:
  - Compute-communicate cycles in typical MPI apps
  - Network is used for a fraction of time
  - 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

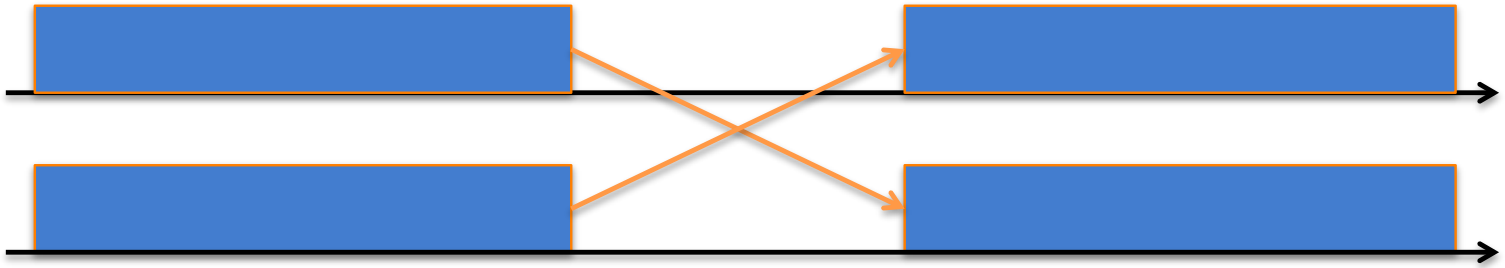
Stencil computation

- 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
  - With Charm++
    - Multiple chunks on each processor
    - Wait time for each chunk overlapped with useful computation for others
    - Communication spread over time

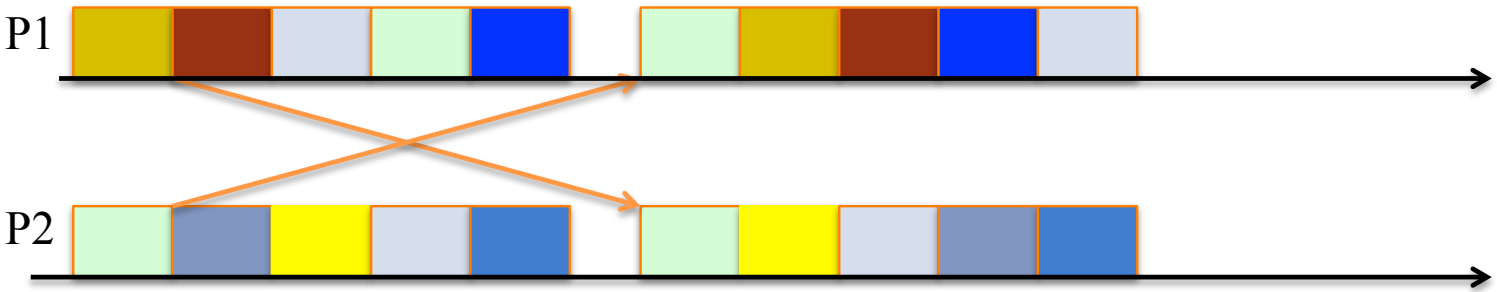
Example

Stencil computation

Stencil in MPI: No overlap among computation and communication

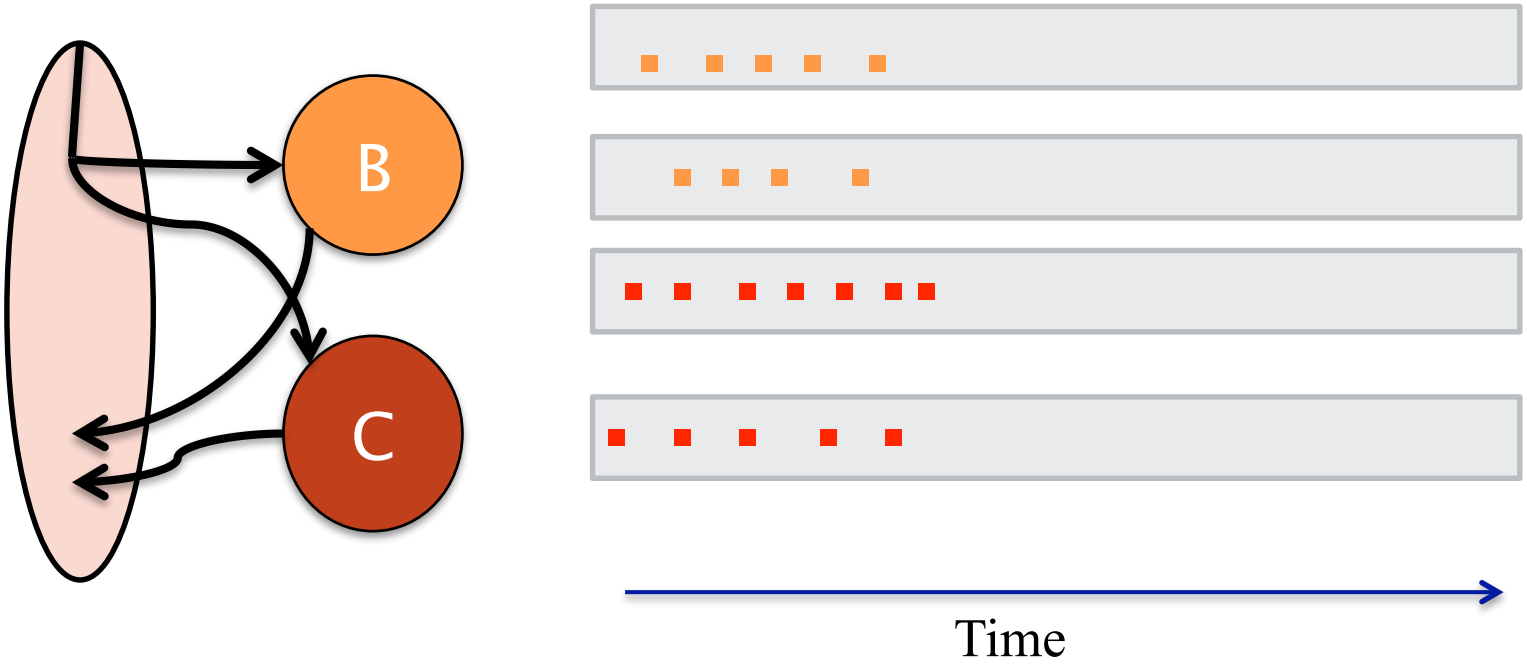


Stencil in Charm: Communication of a chare overlaps with computation of others



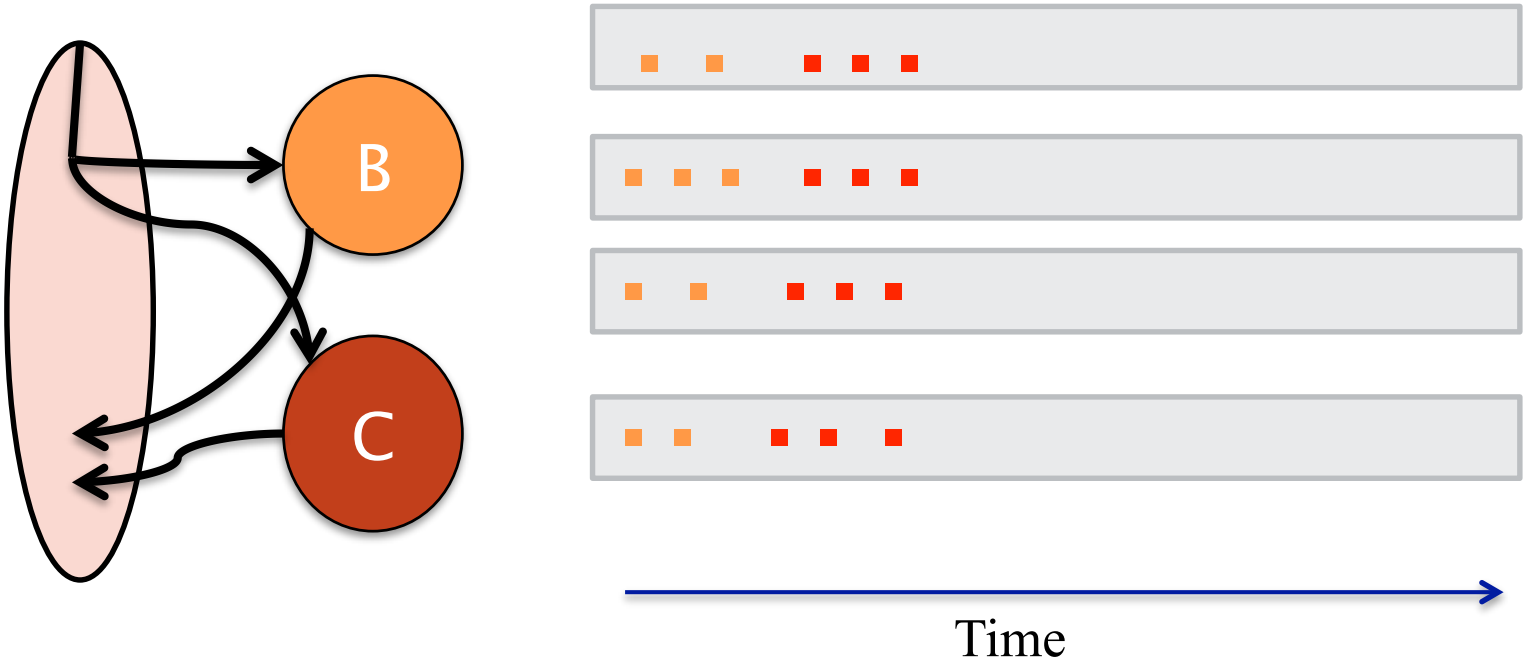
Space Division

An effect of lacking message-driven execution (and virtualization)



Sequentialization

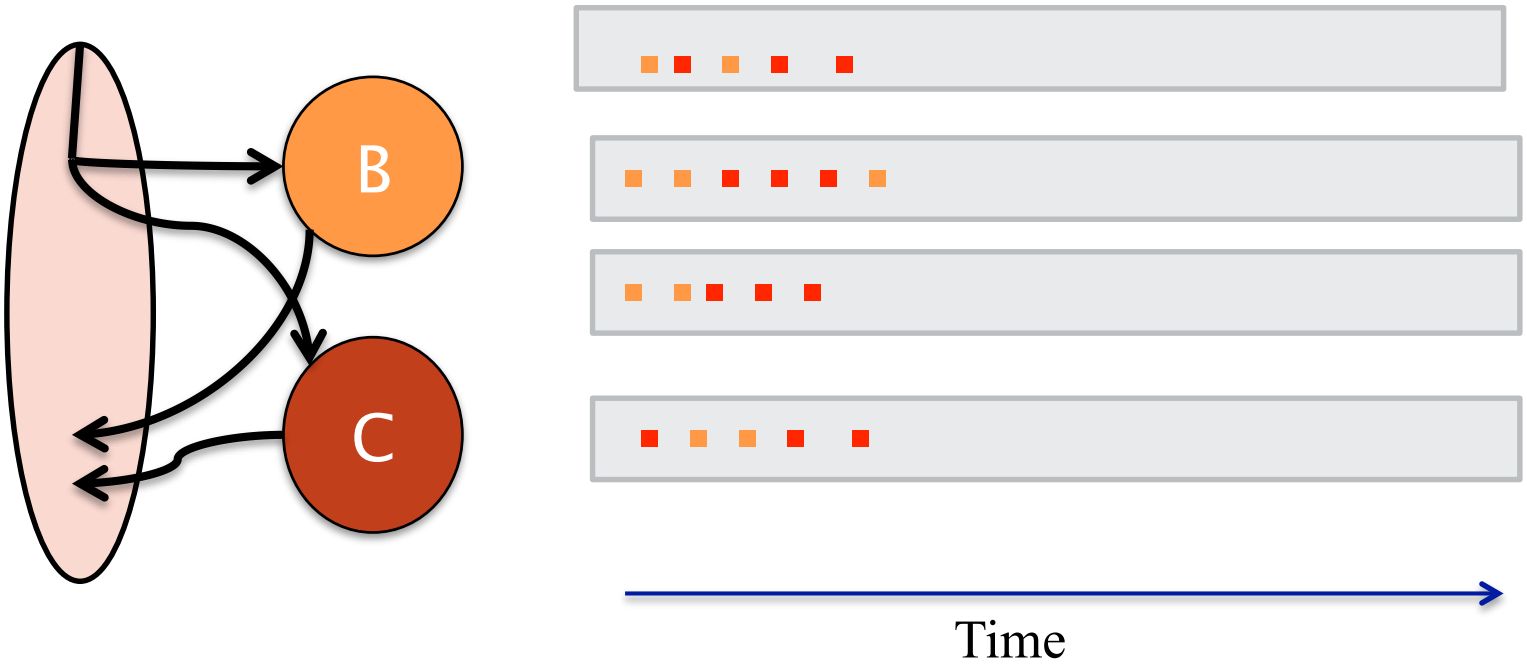
An effect of lacking message-driven execution (and virtualization)



Modularity and Compositionality

Separation of concerns

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



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

CeNAT

Migratability

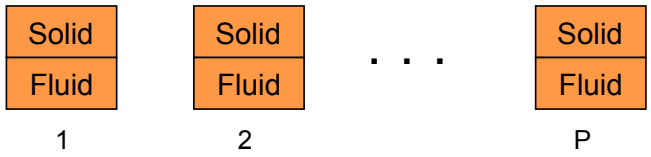
Location independence

- 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, wherever they go
  - This migratability turns out to be a key attribute for empowering an adaptive runtime system

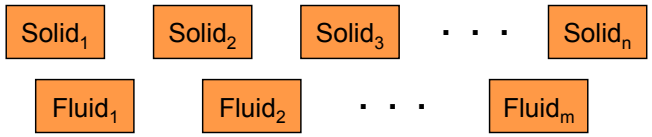
Example

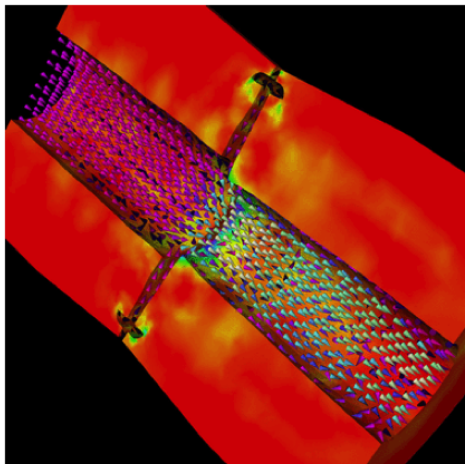
Decomposition independent of number of cores

- Rocket simulation under traditional MPI



- Rocket simulation with migratable objects





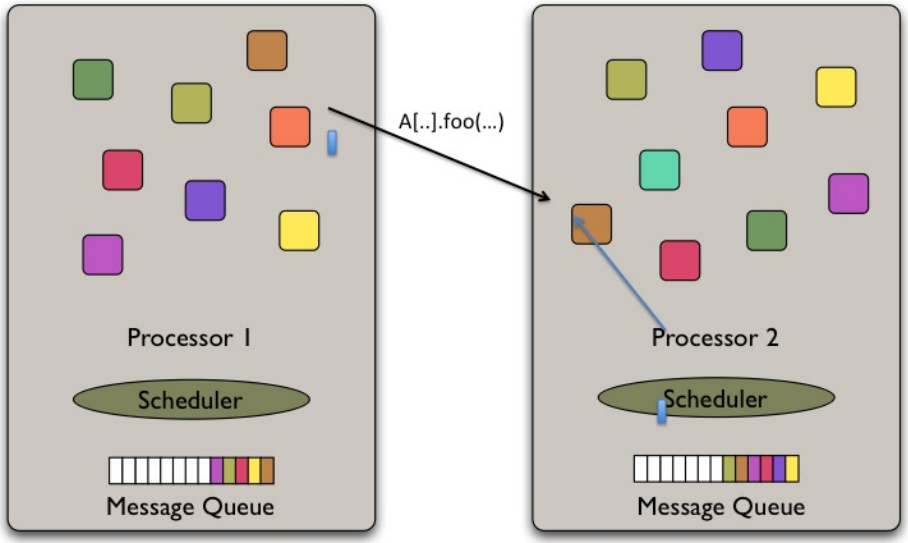
- Benefits: load balance, communication optimizations, modularity

CeNAT

Architectures

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 scratchpad memories



CeNAT

Load Balancing

A major benefit of migratability

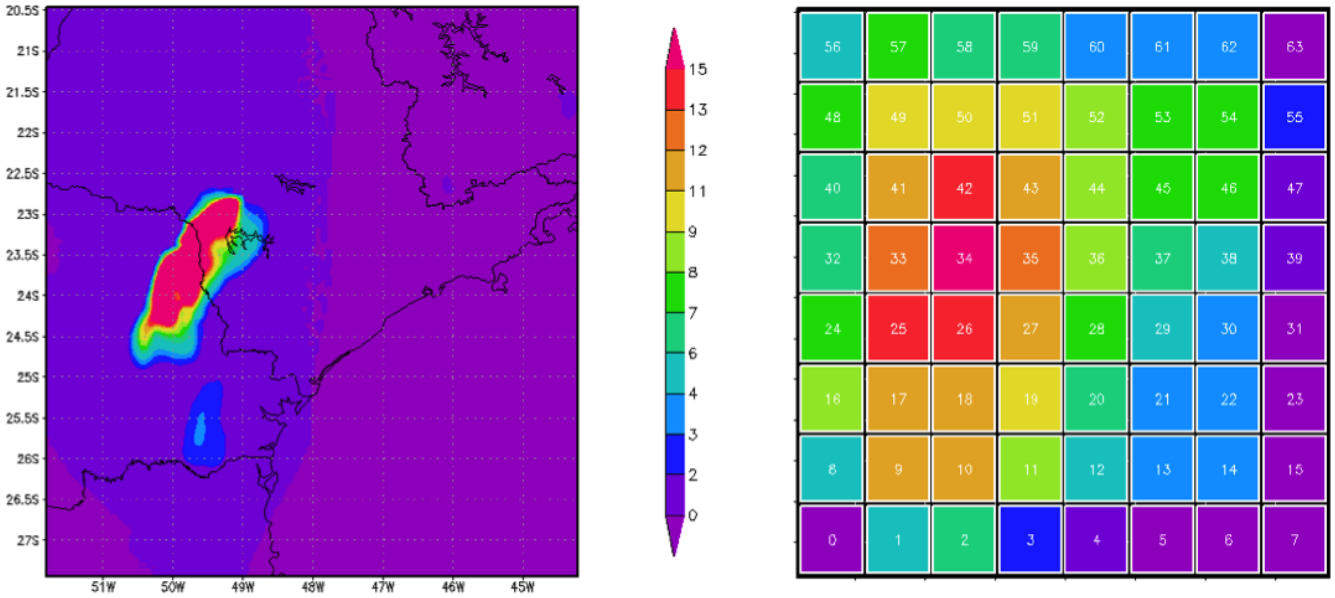
- Static
  - Irregular applications
  - Programmer shouldn't have to figure out ideal mapping
- Dynamic
  - Applications are increasingly using adaptive strategies
  - Abrupt refinements
  - Continuous migration of work (e.g. particles in MD)
- Challenges
  - Performance limited by most overloaded processor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

Migratable objects empower automated load balancing!

Example

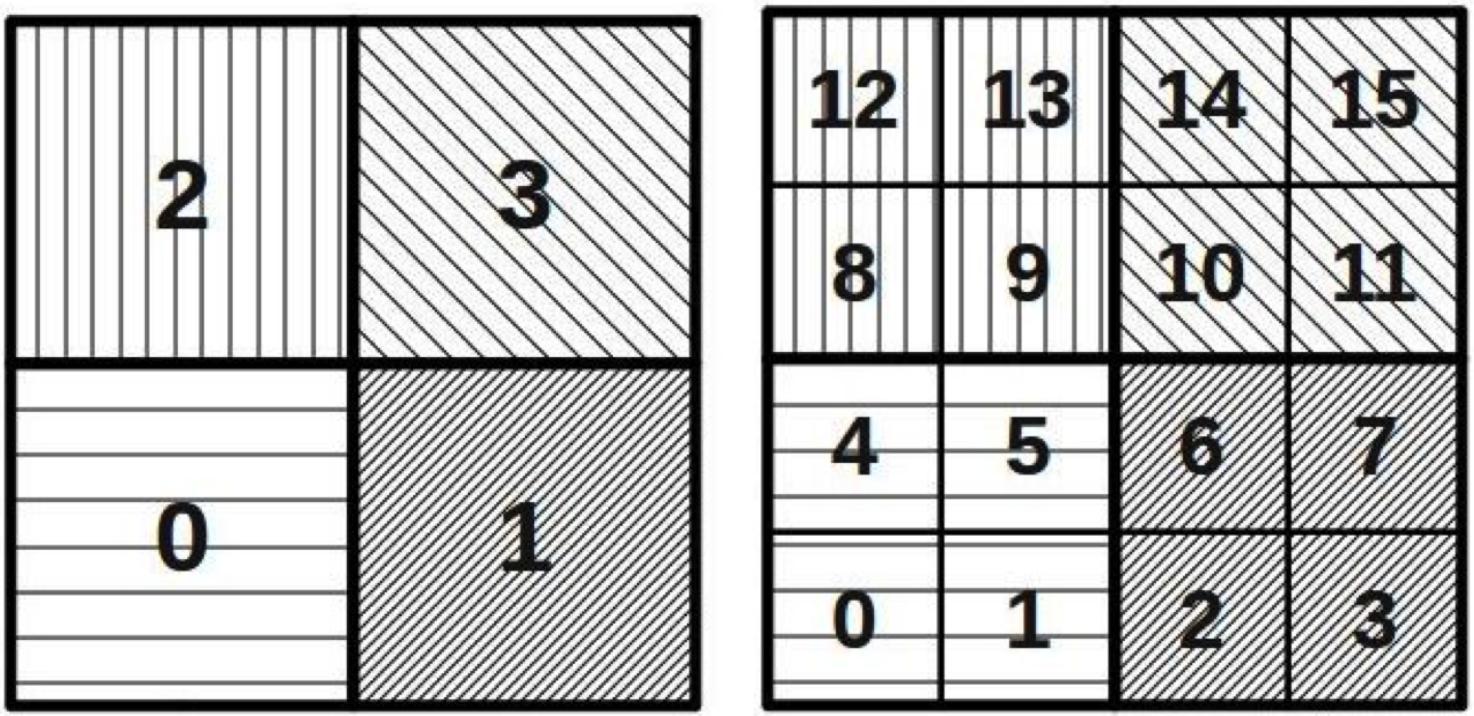
Weather forecasting in BRAMS

- BRAMS: Brazillian weather code (based on RAMS)
- AMPI version (Eduardo Rodrigues, with Dr. Celso Mendes and Prof. Jairo Panetta)



Basic Virtualization of BRAMS

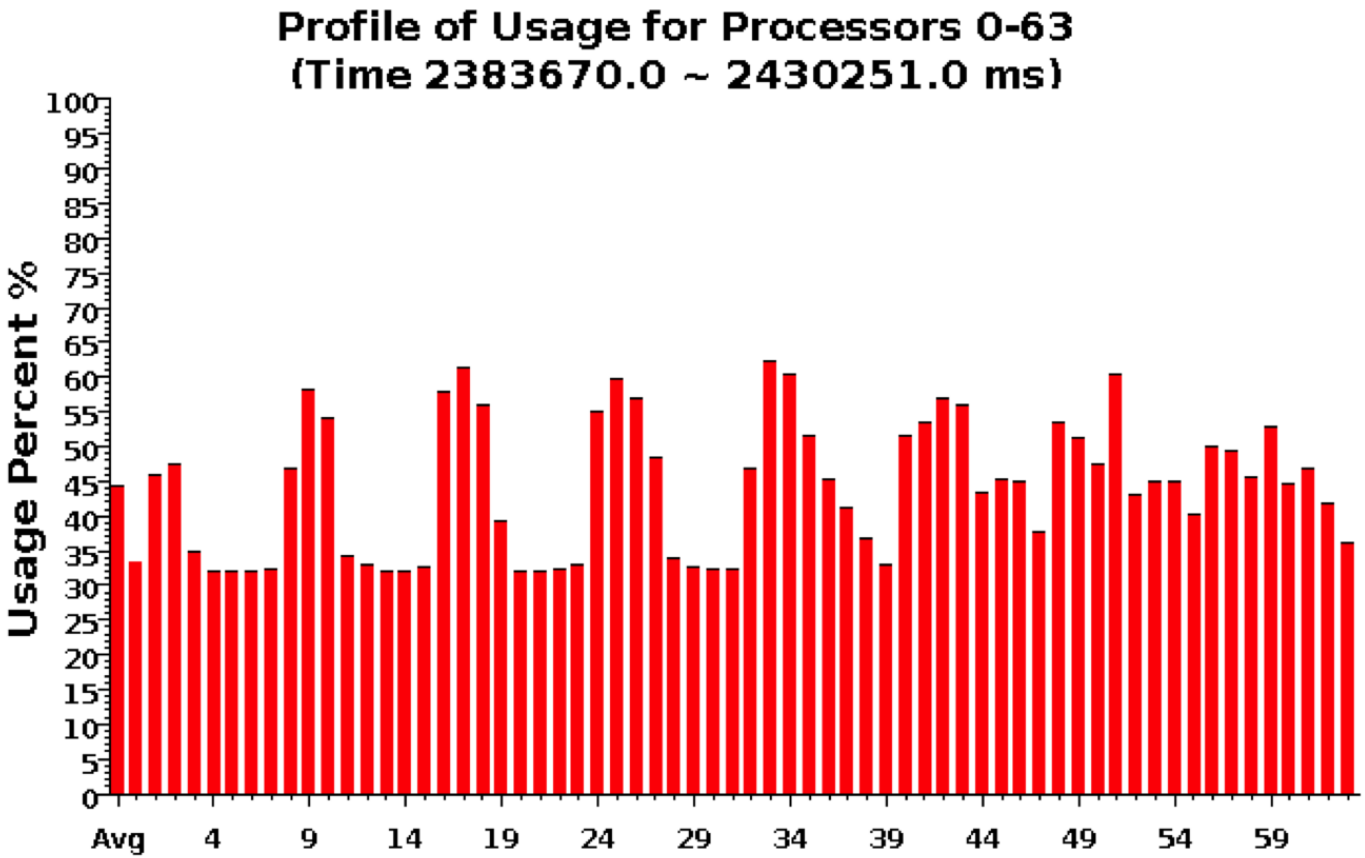
Data partition



CeNAT

Baseline

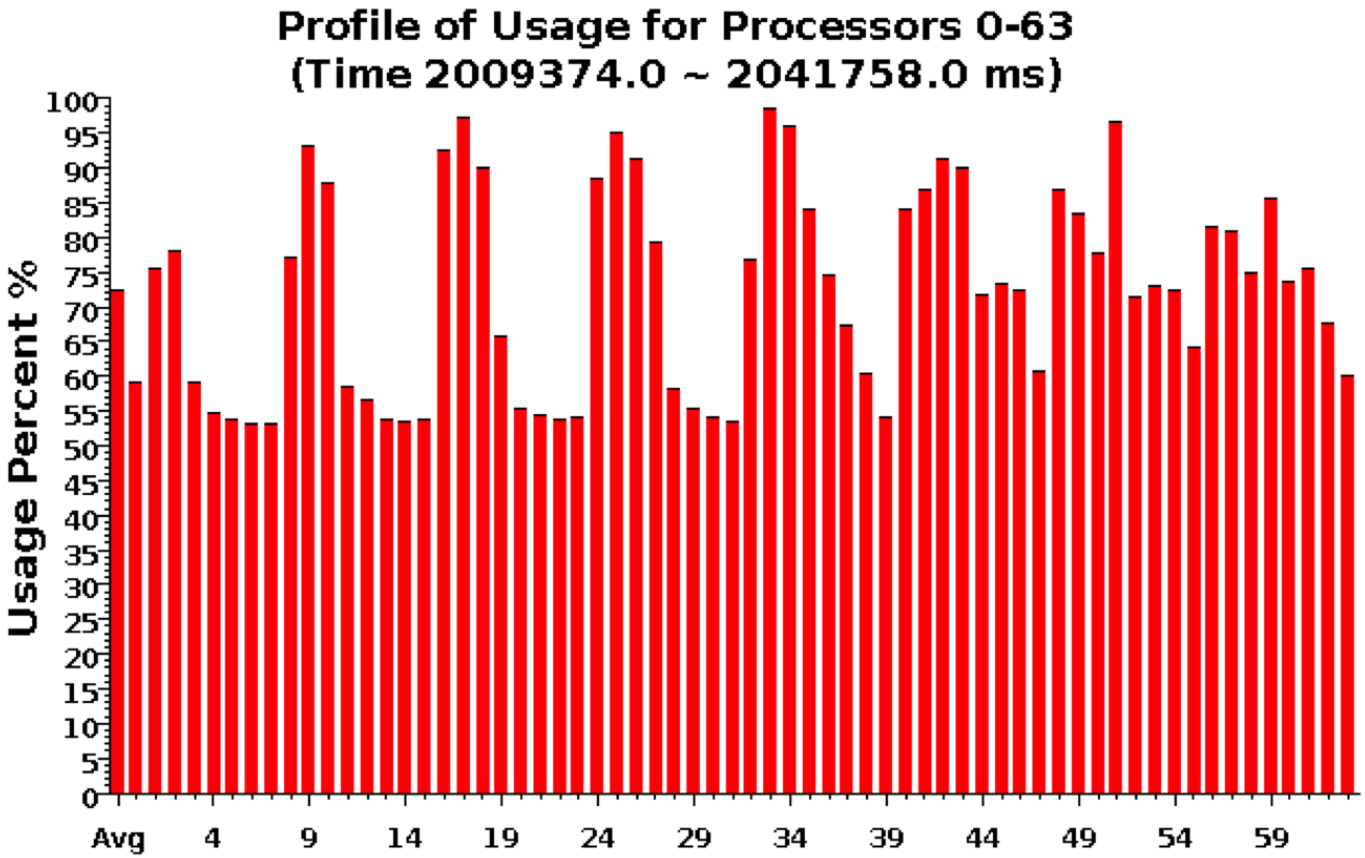
64 objects on 64 processors



CeNAT

Over-decomposition

1024 objects on 64 processors



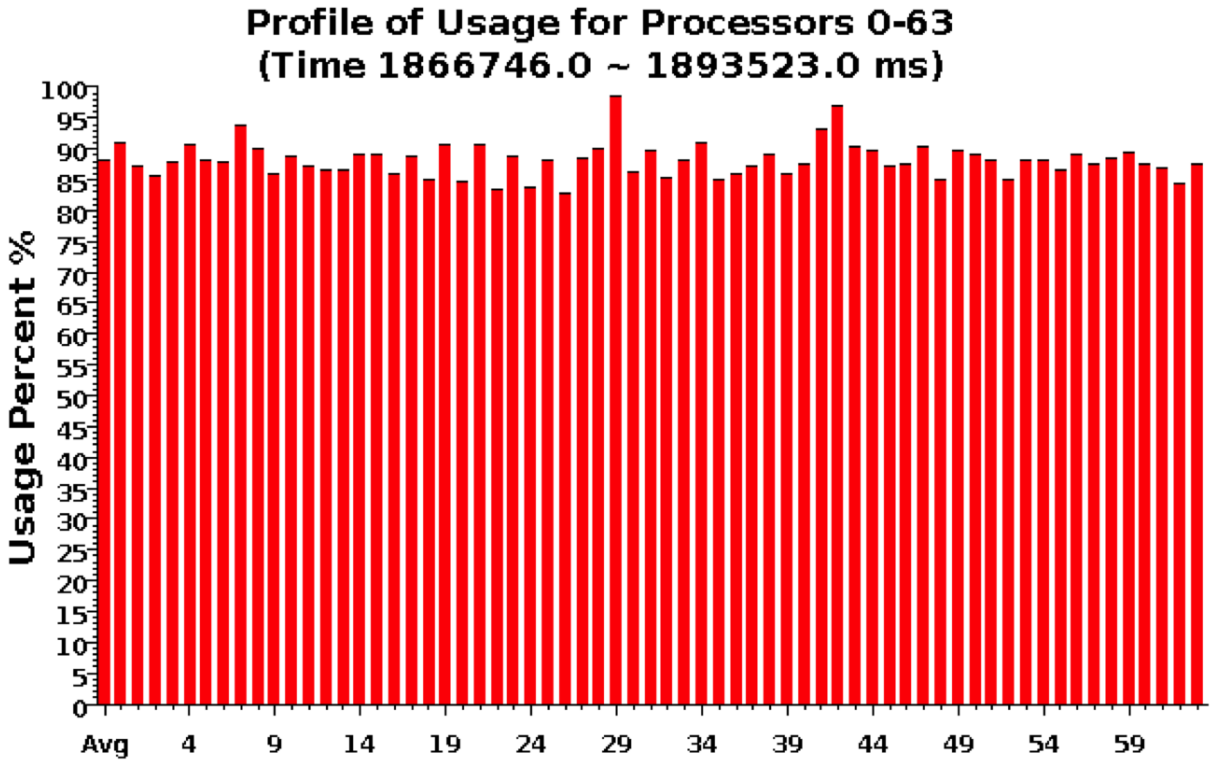
Benefits from communication/computation overlap

CeNAT

With Load Balancing

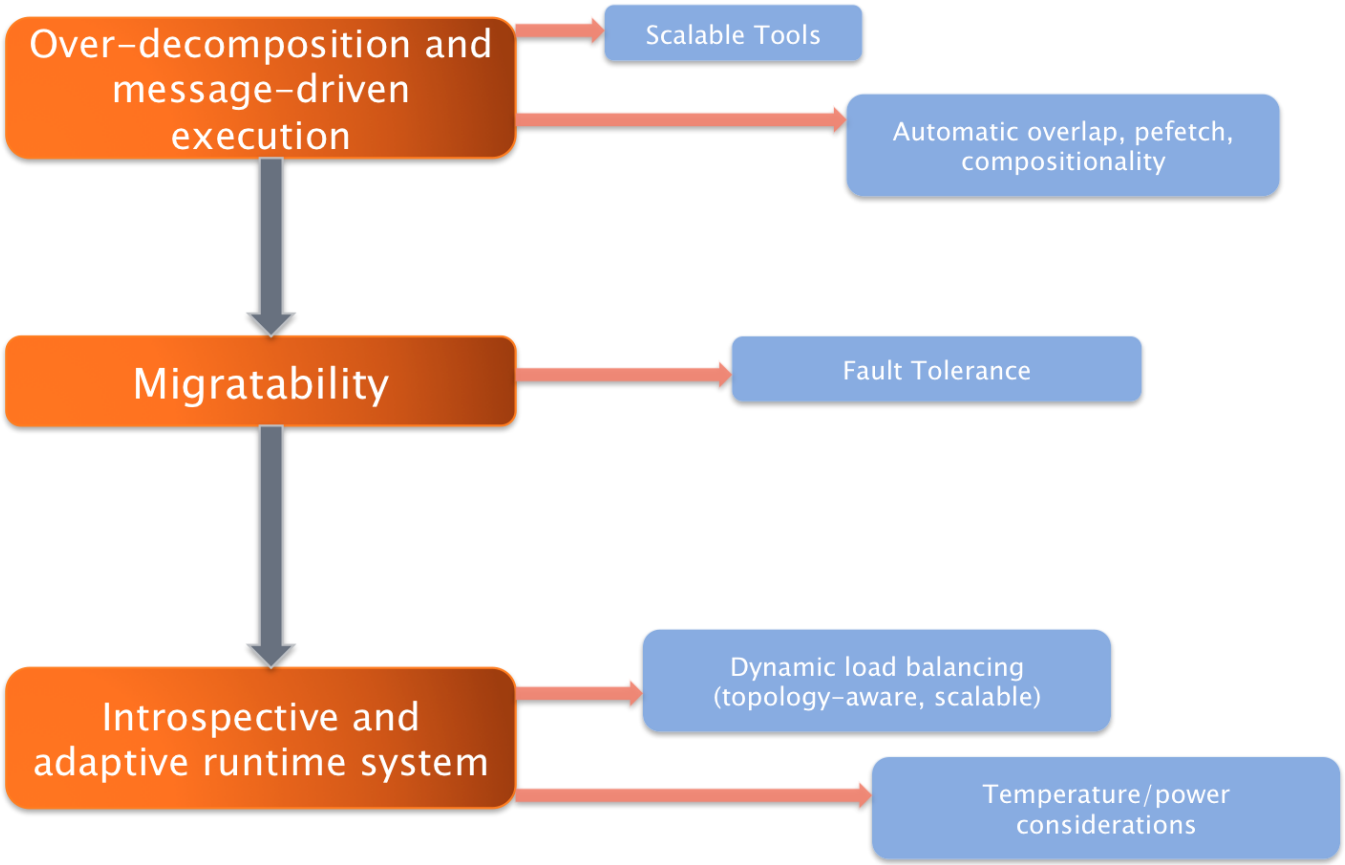
1024 objects on 64 processors

- No overdecomposition (64 threads): 4988 sec
- Overdecomposition into 1024 threads: 3713 sec
- Load balancing (1024 threads): 3367 sec



CeNAT

Summary of Benefits



CeNAT

The Charm++ Programming Language

History

University of Illinois at Urbana-Champaign, early 1990s

Prof. Laxmikant V. Kalé

Prof. Klaus Schulten

Parallel Programming Laboratory (Kalé)

Theoretical and Computational Biophysics Group (Schulten)

CeNAT

Parallel Programming Galore

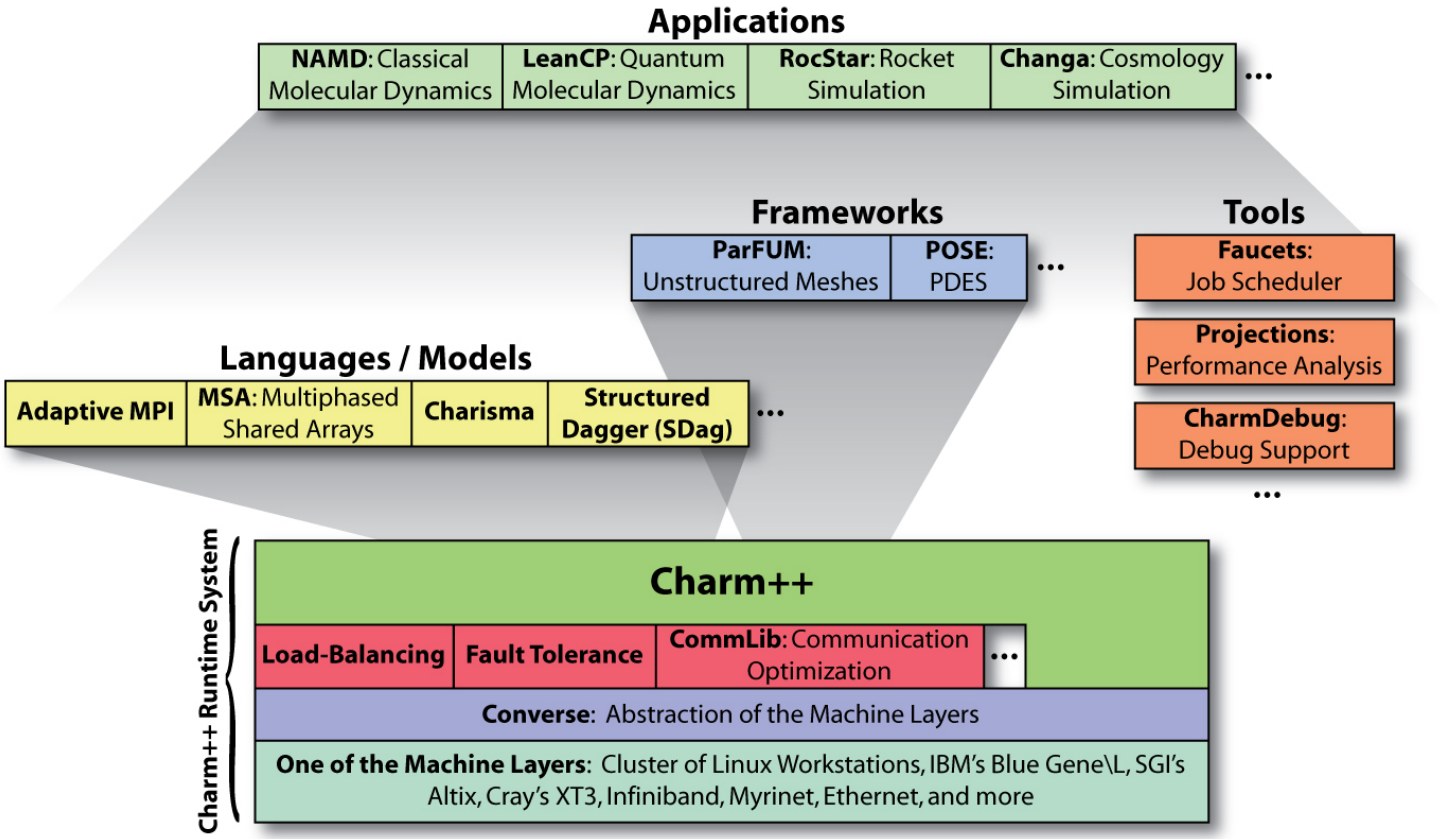
List of Parallel Programming Languages in the 90s

"C* in C	CUMULVS	Java RMI	P-RIO	Quake
ABCPL	DAGGER	javaPG	P3L	Quark
ACE	DAPPLE	JAVAR	P4-Linda	Quick Threads
ACT++	Data Parallel C	JavaSpaces	Pablo	Sage++
ADDAP	DC++	JIDL	PADE	SAM
Adl	DCE++	Joyce	PADRE	SCANDAL
Adsmith	DDD	Karma	Panda	SCHEDULE
AFAPI	DICE	Khoros	Papers	SciTL
ALWAN	DIPC	KOAN/Fortran-S	Para++	SDDA
AM	Distributed Smalltalk	LAM	Paradigm	SHMEM
AMDC	DOLIB	Legion	Parafrase2	SIMPLE
Amoeba	DOME	Lilac	Paralation	Sina
AppLeS	DOSMOS	Linda	Parallaxis	SISAL
ARTS	DRL	LiPS	Parallel Haskell	SMI
Athapascan-0b	DSM-Threads	Locust	Parallel-C++	SONiC
Aurora	Ease	Lparx	ParC	Split-C
Automap	ECO	Lucid	ParLib++	SR
bb_threads	Eiffel	Maisie	ParLin	Sthreads
Blaze	Emerald	Manifold	Parlog	Strand
BlockComm	EPL	Mentat	Parmacs	SUIF
BSP	Excalibur	Meta Chaos	Parti	SuperPascal
C*	Express	Midway	pC	Synergy
C**	Falcon	Millipede	pC++	TCGMSG
C4	Filaments	Mirage	PCN	Telegraphos
CarlOS	FLASH	Modula-2*	PCP:	The FORCE
Cashmere	FM	Modula-P	PCU	Threads.h++
CC++	Fork	MOSIX	PEACE	TRAPPER
Charlotte	Fortran-M	MpC	PENNY	TreadMarks
Charm	FX	MPC++	PET	UC
Charm++	GA	MPI	PETSc	uC++
Chu	GAMMA	Multipol	PH	UNITY
Cid	Glenda	Munin	Phosphorus	V
Cilk	GLU	Nano-Threads	POET	ViC*
CM-Fortran	GUARD	NESL	Polaris	Visifold V-NUS
Code	HAsL	NetClasses++	POOL-T	VPE
Concurrent ML	HORUS	Nexus	POOMA	Win32 threads
Converse	HPC	Nimrod	POSYBL	WinPar
COOL	HPC++	NOW	PRESTO	WWWinda
CORRELATE	HPF	Objective Linda	Prospero	XENOOPS
CparPar	IMPACT	Occam	Proteus	XPC
CPS	ISETL-Linda	Omega	PSDM	Zounds
CRL	ISIS	OOF90	PSI	ZPL
CSP	JADA	Orca	PVM	
Cthreads	JADE	P++	QPC++	

"Patterns for Parallel Programming" (Tim Mattson, et al, 2004)

Charm++ Runtime System

A whole programming ecosystem

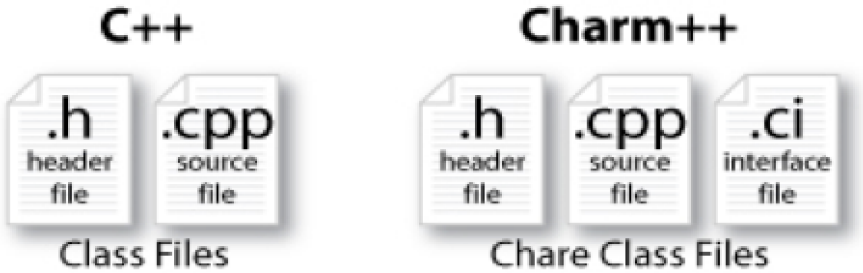


CeNAT

Charm++ File Structure

Header, source, and interface files

- 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 create some additional chares

Creating a Chare

Interacting object on a distributed system

- 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

Representative of a real object

- 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

More than finishing execution

- 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 required 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)

Example

Chare creation, `.ci` file

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

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

Example

Chare creation, .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

Beyond two-sided communication

- 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

Beyond FIFO channels

- 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;
}
```

Example

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
```

Example

Asynchronous `.ci` file

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

Exercise 1.3

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

STL supported

- 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 ...
}
```

Readonlys

Constants on a distributed system

- A *readonly* is a global (within a module) read-only variable that can only be written to in the `mainchare`'s constructor
- Can then be read (**not written!**) by any chare in the module
- It is declared in the `.ci` file:

```
readonly <type> <name>;
readonly CProxy_Main mainProxy;
readonly int numChares;
```

- And defined the the `.C` file:

```
<type> <name>;
CProxy_Main mainProxy;
int numChares;
```

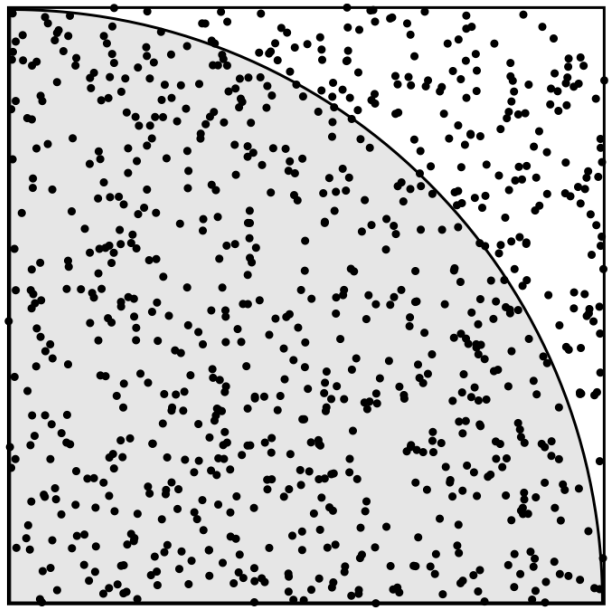
- And set in the `mainchare`'s constructor

```
MyChare::MyChare(CkArgMsg *m) {
  mainProxy = thisProxy;
  numChares = 10;
}
```

Exercise 1.4

Computing π

Use a Monte Carlo method to compute π:



Complete the code in directory:

`code/pi/skeleton`

of the class repository:

```
git clone https://github.com/emenesesrojas/parallel_objects.git
```

Do not forget to define variable `CHARMDIR` in the Makefile

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

Types of collections

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

Example

Chare Array Hello World

```
mainmodule arr {


  mainchare Main {
    entry Main(CkArgMsg*);
  }

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

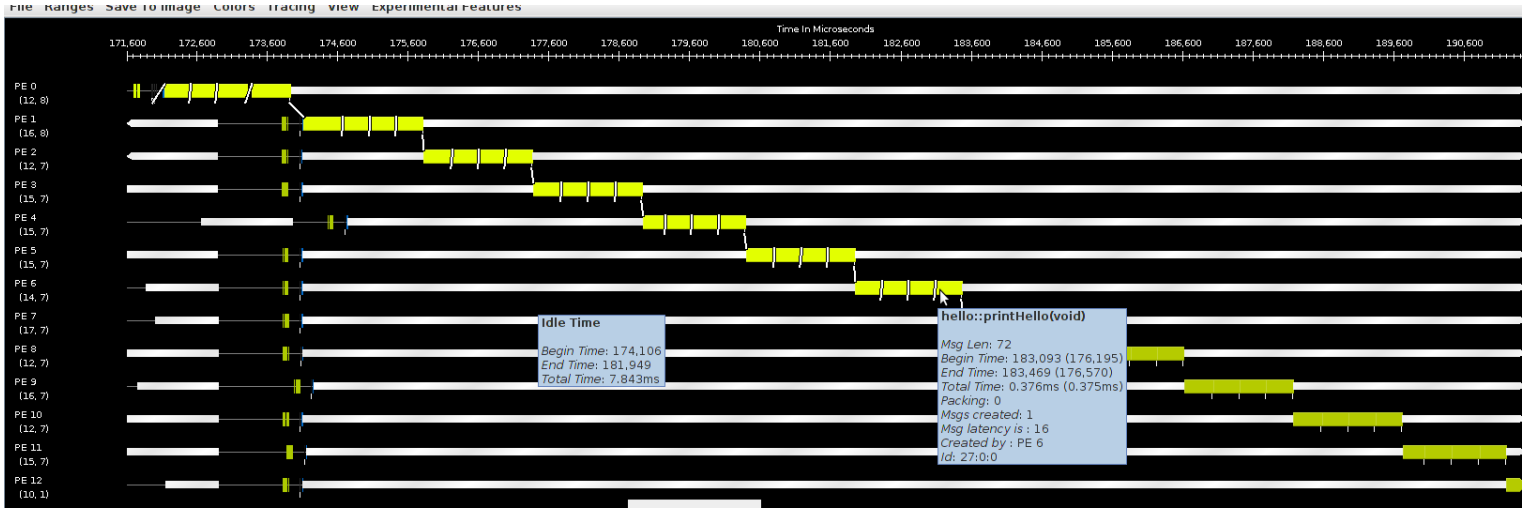
Example

Chare array Hello World

```
#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"
```

Example

Hello World array Projections timeline view



Run on BG/Q 16 Nodes, mode c16, 1024 elements (4 per process)

Declaring a Chare Array

Dimensionality and functionality

`.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*) { }
  // ... entry methods ...
};
```


CeNAT

Constructing a Chare Array

Elements are distributed on nodes of the system

- 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

Pretty much like this

- 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);
  }
};
```

Collections of Objects

Runtime service

- System knows how to 'find' objects efficiently: $(\textit{collection}, \textit{index}) \rightarrow \textit{processor}$
- 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

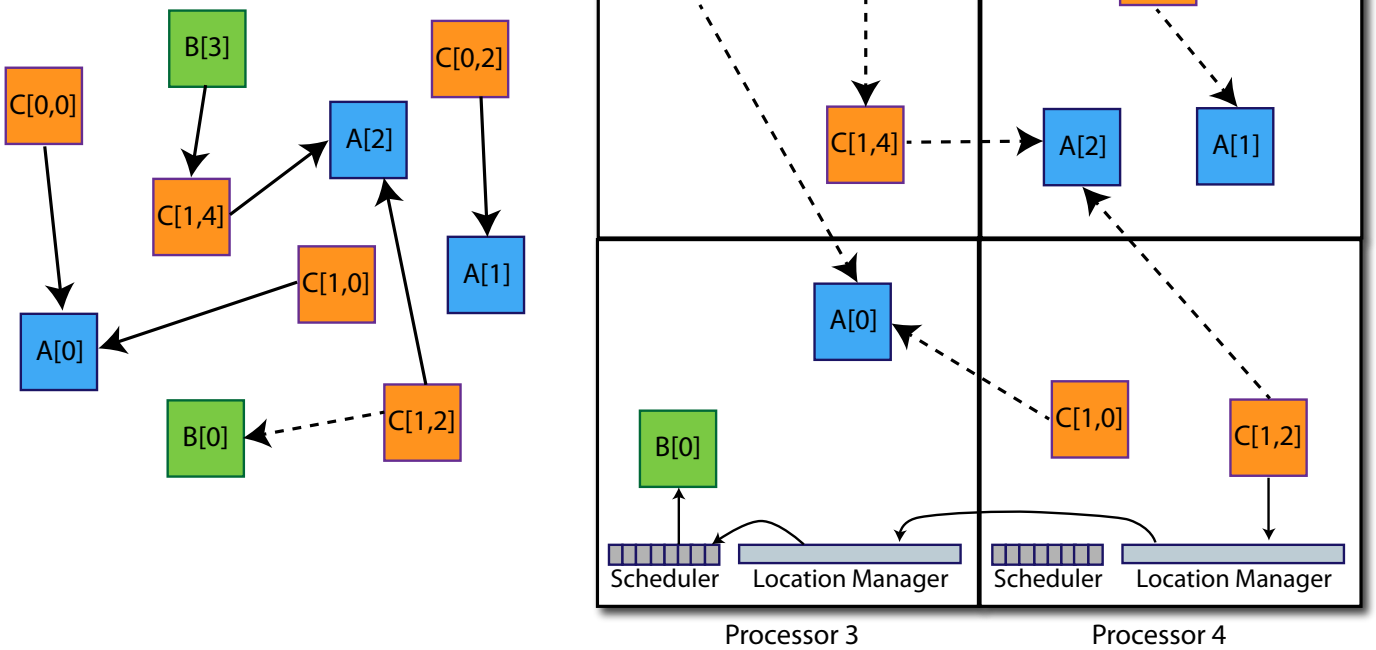
Collections of Objects

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

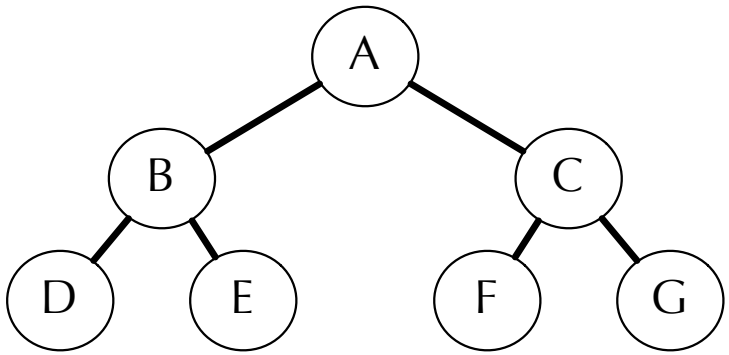
Interacting collections



Collective Communication Operations

Efficiency and expressivity

- 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



CeNAT

Broadcast

One-to-many

- 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

Many-to-one

- 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

Example

Reduction

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

Example

Reduction

```
#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.
```

Exercise 1.5

Key-value store

Implement a key-value store using a 1D chare array:

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Complete the code in directory:

`code/keyvalue/skeleton`

of the class repository:

```
git clone https://github.com/emenesesrojas/parallel_objects.git
```

Do not forget to define variable `CHARMDIR` in the Makefile

Concluding Remarks

- Parallel objects bring about a modular and composable parallel programming framework
- Two major benefits of over-decomposition are:
  - Overlap of communication and computation
  - Load balancing
- Chare arrays are the main object collection in Charm++

Thank You!
Q&A