ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

Parallel Computing w/ Charm++

November 18, 2015

Lecture 28

Quote of the Day

“Money doesn't buy happiness, yet everybody would like to find out if it is true.”

-- Stefan Kisielewski (1911 – 1991)

Polish writer, publicist, composer and politician

The Story Behind "The Quote of the Day"
[or how I became interested in this topic]

"It is a good thing for an uneducated man to read books of quotations."

-- Sir Winston Churchill

1874 - 1965

Before We Get Started

- Issues covered last time:
  - CUDA, OpenMP, MPI: putting things in perspective
  - Other parallel programming models, quick overview
    - TBB
    - C++11
    - Cilk
    - Chapel

- Today's topics
  - Charm++
  - ME759 wrap up

- Other issues:
  - Today is the last lecture of the semester
  - HW09 due tonight at 11:59 PM
  - HW10 uploaded tonight
  - Recall that you can drop two lowest score assignments
  - Final Project Proposal: if you don't hear from me by Friday, it means that your proposal was fine
  - Second (and last) exam: coming up on 11/23 (Monday) at 7:15 PM (Room: 1610EH)
    - Review on 11/23 in 1610EH during regular lecture hours

- Charm++ material from Professor Kale
  - Modified here and there
  - Mistakes in the slides mostly likely traceable to my changes

- Charm++ started around 1993 or so
  - Professor Kale in his lab at University of Illinois at Urbana/Champaign

Challenges in HPC. Where Charm++ Fits in the Picture

- Applications are getting more sophisticated
  - Multi-scale, multi-module, multi-physics
- Strong scaling needs from apps
  - Working on 100s of nodes is run of the mill in HPC
  - Load imbalance emerges as a big issue for some apps
- Design philosophy of Charm++
  - Do not attempt full automation;
    - That is, don't fully rely on compiler and/or runtime
  - Yet don't place burden on app-developer's shoulder
  - Take the middle road
    - Seek a good division of labor between the system and app developers

What is Charm++?

- Charm++ is a generalized approach to writing parallel programs
  - An alternative to the likes of MPI, Chapel, UPC, etc.
- Charm++, three facets
  - A style of writing parallel programs
  - An ecosystem that facilitates the act of writing parallel programs
    - Debugger, profiler, ability to define own load balancing, etc.
  - A runtime system
- Three design principles (the tenets of Charm++)
    - Overdecomposition
    - Migratability
    - Asynchrony

Overdecomposition

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

- Why do this?
  - Recall the GPU computing strategy
    - You want to have many warps in flight to find one that is ready to go
  - Central idea: oversubscription of the hardware
    - Hide memory latency w/ useful execution
  - This oversubscription idea is a general tenet
    - Embraced in setting up a programming model but also by a person writing his/her own application

Migratability

- Make the work and data units on previous slide migratable at runtime
  - That is, the programmer or runtime can move them from execution unit (PE, from processing element) to execution unit
    - From PE to PE, that is
- Consequences for the app-developer
  - Communication must now be addressed to logical units with global names, not to physical processors
  - But this is a good thing
- Consequences for the runtime system (RTS)
  - Must keep track of where each unit is
  - Naming and location management

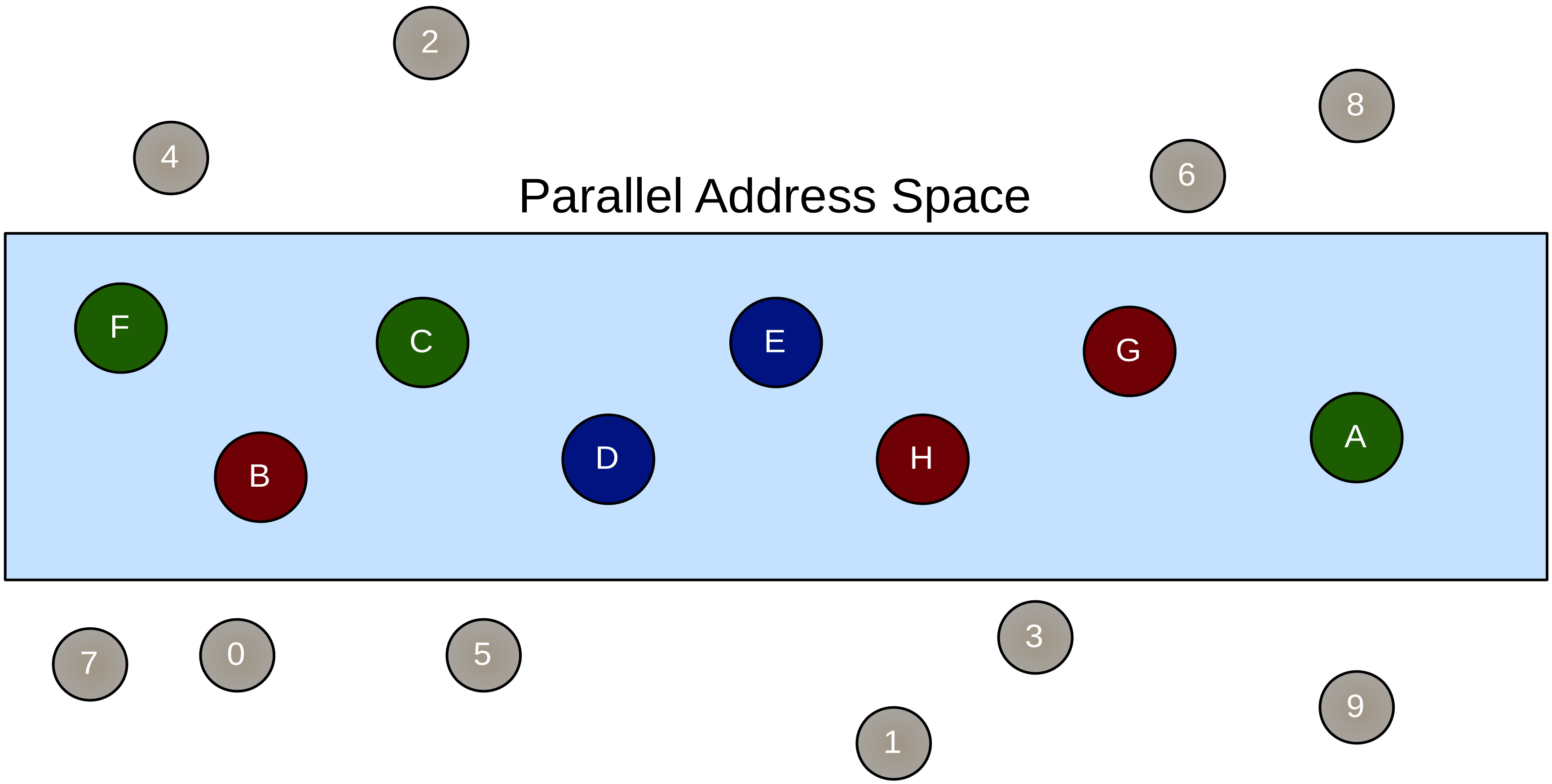
Asynchrony: Message-Driven Execution

- Here we are:
  - We have multiple work units (“things to do”) on each PE
  - Work units can address/invoke each other via logical names
- Scheduling question: What sequence should the work units execute in?
  - One answer: let the programmer sequence them
    - “sequence” - like in specifying the order of their execution
    - The common way
  - Let the RTS control this
    - Possible strategy:
      - Let the work-unit that happens to have the necessary data (“message”) execute next
    - Consequence:
      - Programmer does not specify what executes next, but can influence order via priorities

Realization of This Model in Charm++

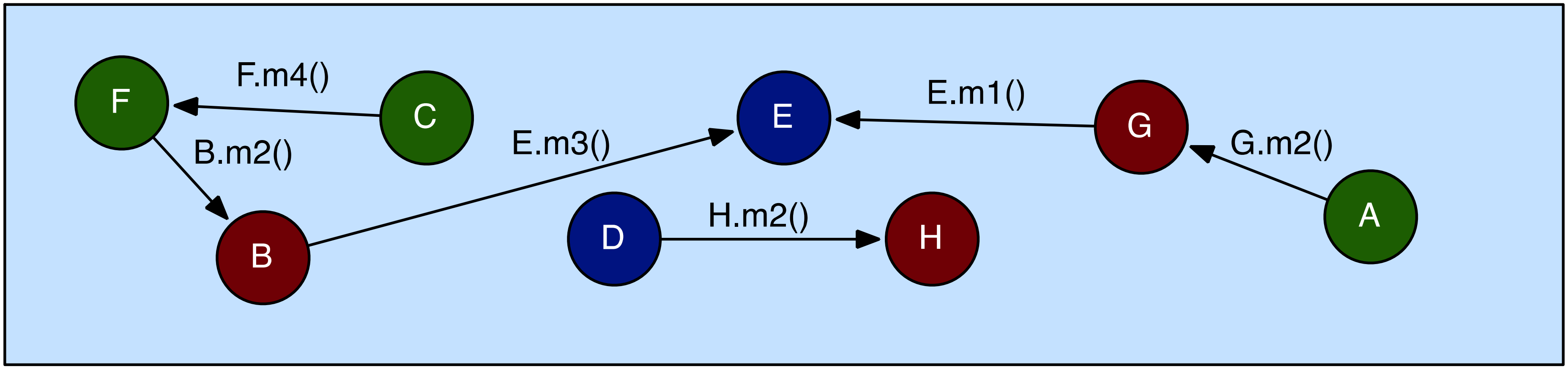
- Overdecomposed entities: chares
  - Chares are C++ objects
    - They have methods designated as “entry” methods
      - These special “entry” methods can be invoked asynchronously by remote chares
  - Chares can be organized into indexed collections
    - Each collection may have its own indexing scheme
      - 1D, 2D,.., 7D
      - Sparse
      - Bitvector or string as an index
  - Chares communicate via asynchronous method invocations
    - `A[i].foo(…);`
      - `A` is the name of a collection, `i` is the index of the particular chare.

Overdecomposed Objects



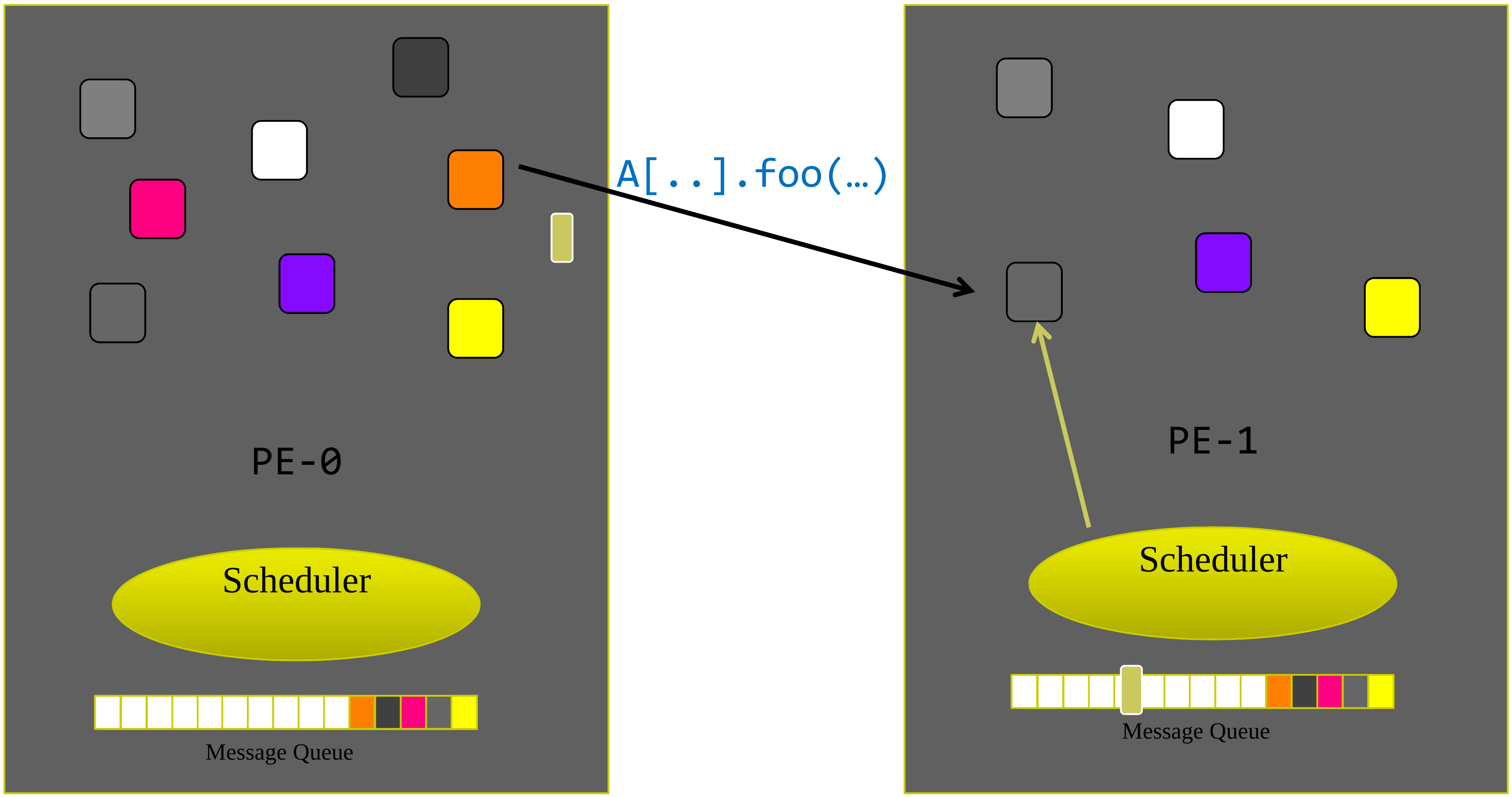
Message-Driven Execution

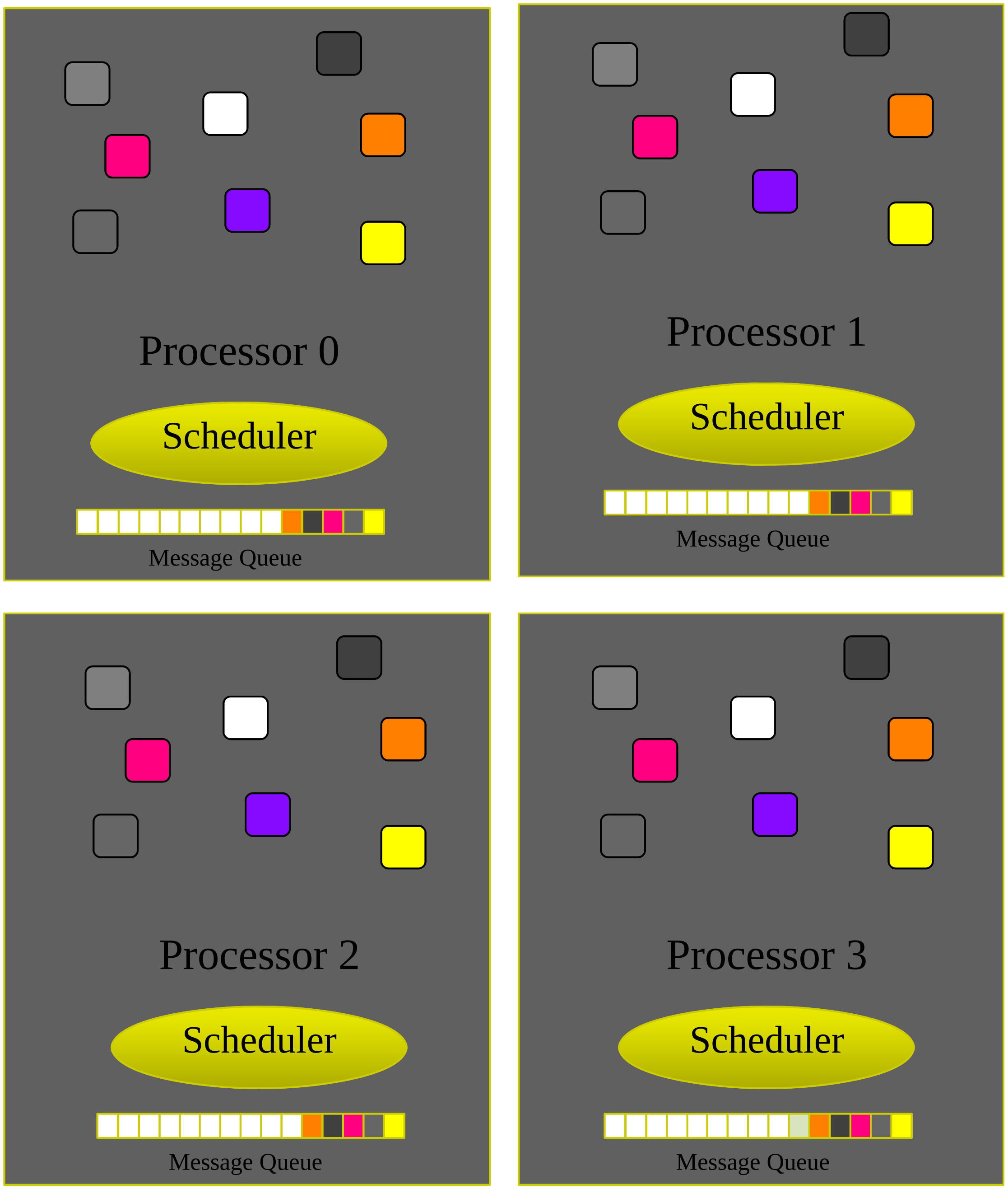
Parallel Address Space

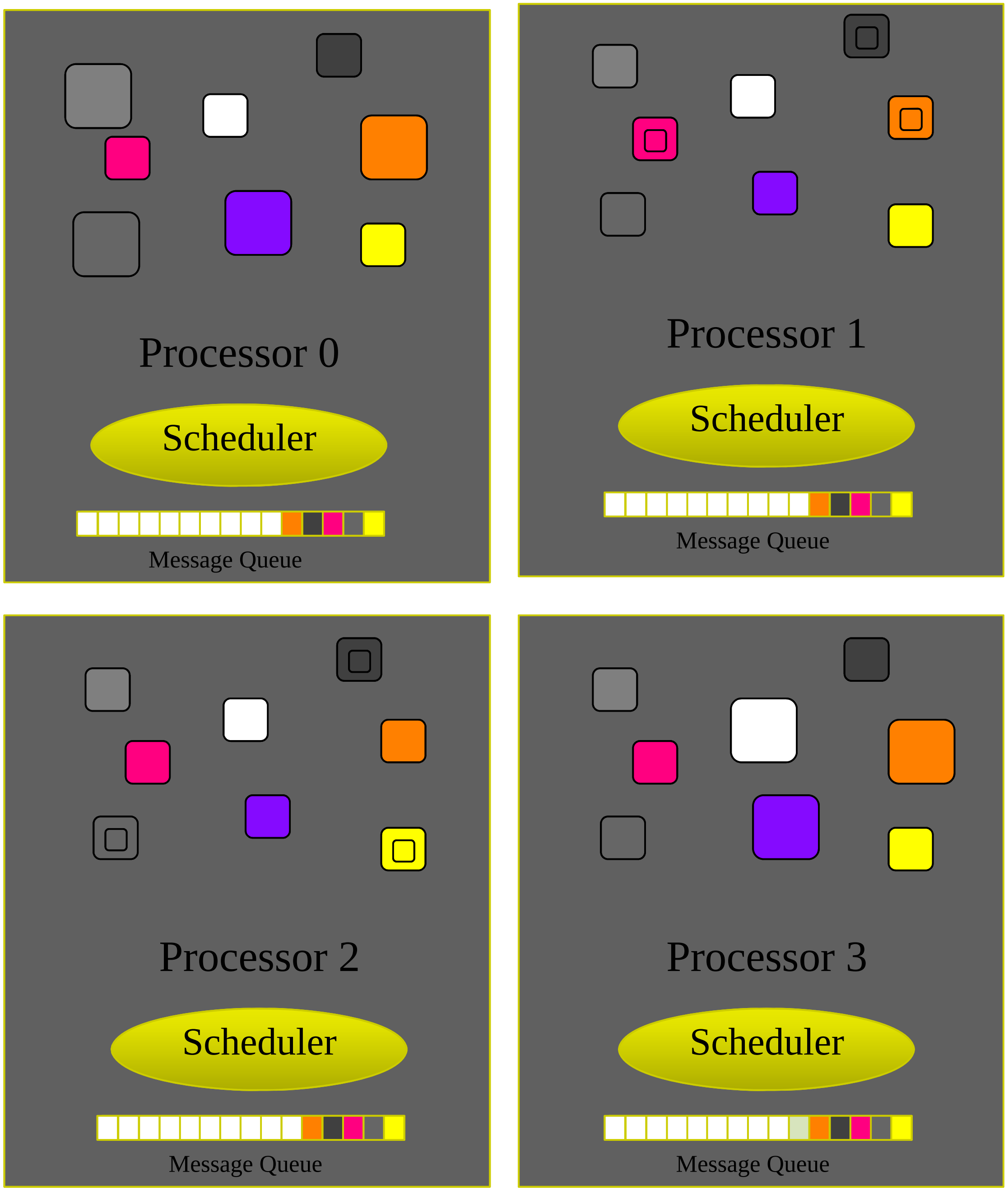


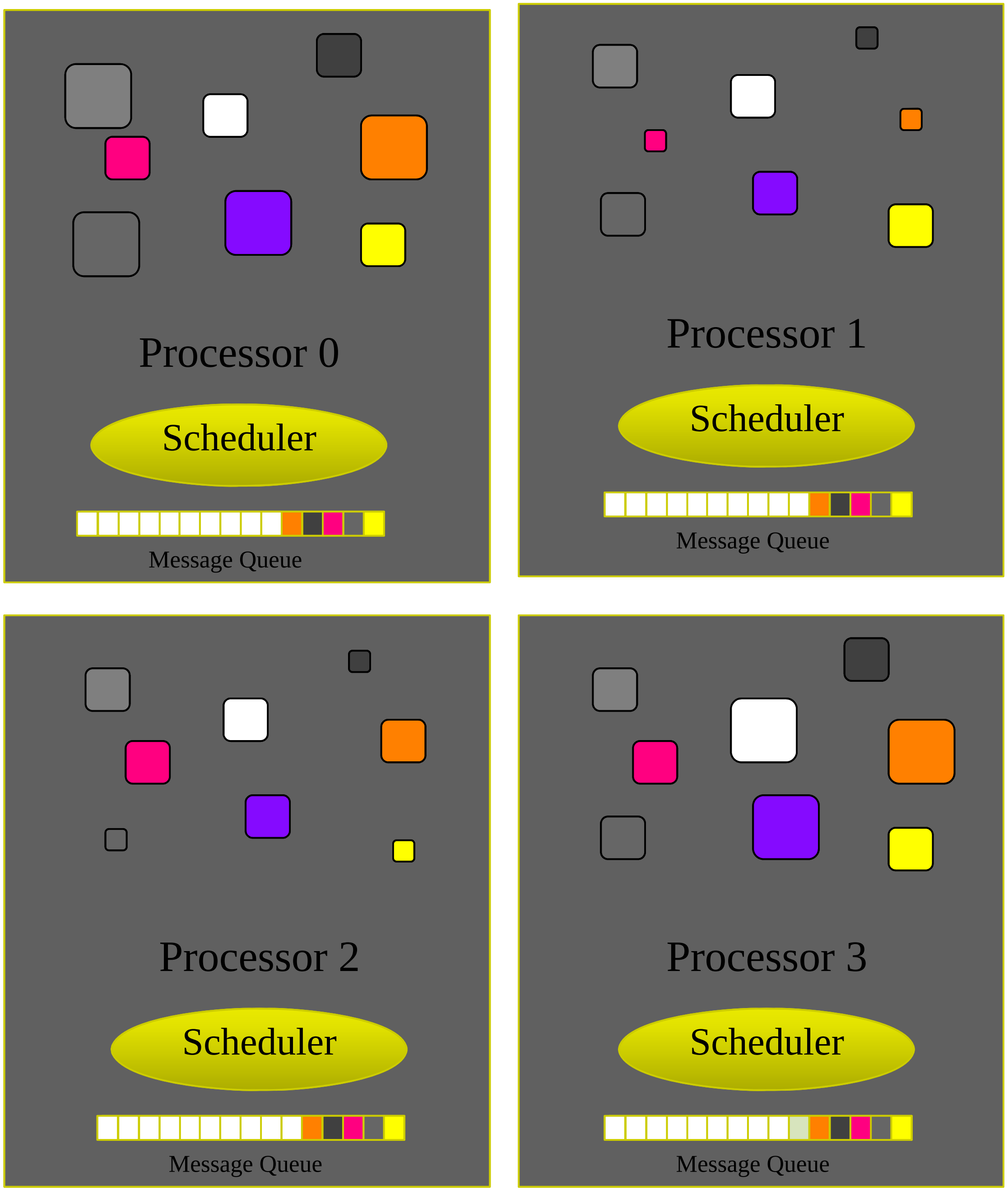
- Certain member functions of certain classes are globally visible
- Invocation of a member function may lead to communication
- "message driven execution" – like in "executing by calling a method on a distant chare and passing arguments to the method call"
  - Data stored in two places:
    - The chare that provides the method that is called stores data
    - The chare the makes the call uses some arguments in the call and the arguments can store data

Message-driven Execution

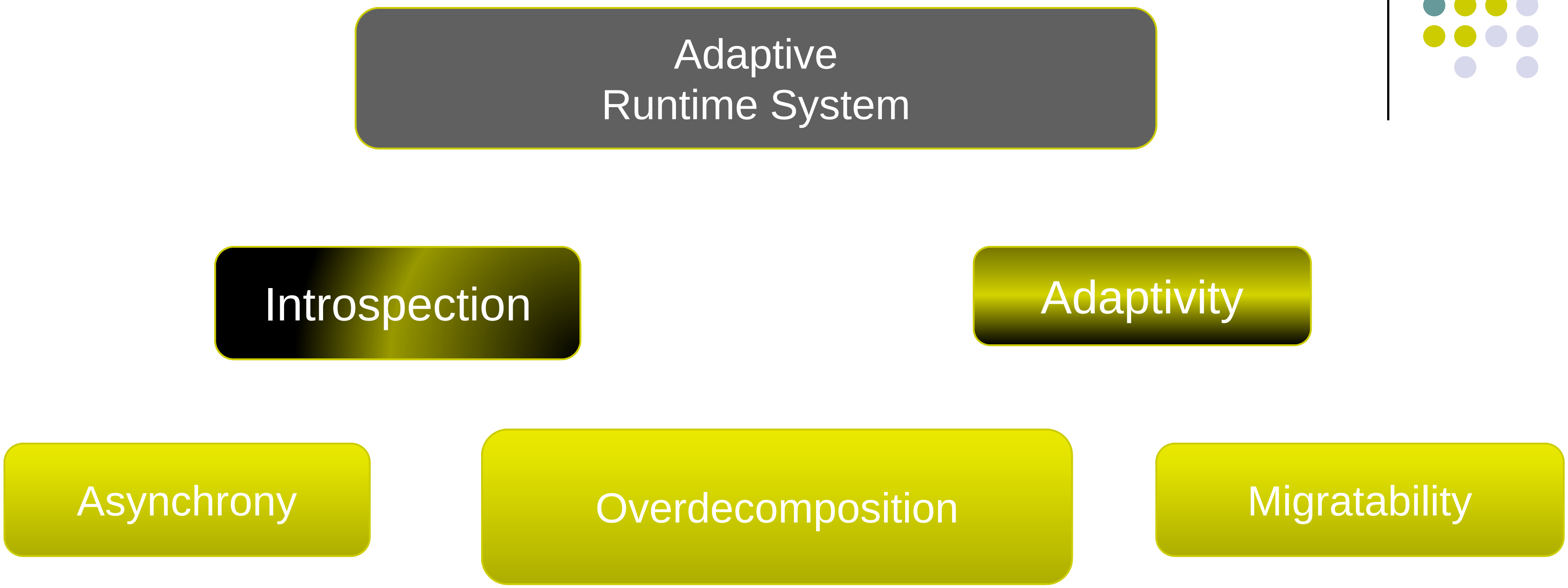



Processor 0
Scheduler
Message Queue
Processor 1
Scheduler
Message Queue
Processor 2
Scheduler
Message Queue
Processor 3
Scheduler
Message Queue


Processor 0
Scheduler
Message Queue
Processor 1
Scheduler
Message Queue
Processor 2
Scheduler
Message Queue
Processor 3
Scheduler
Message Queue

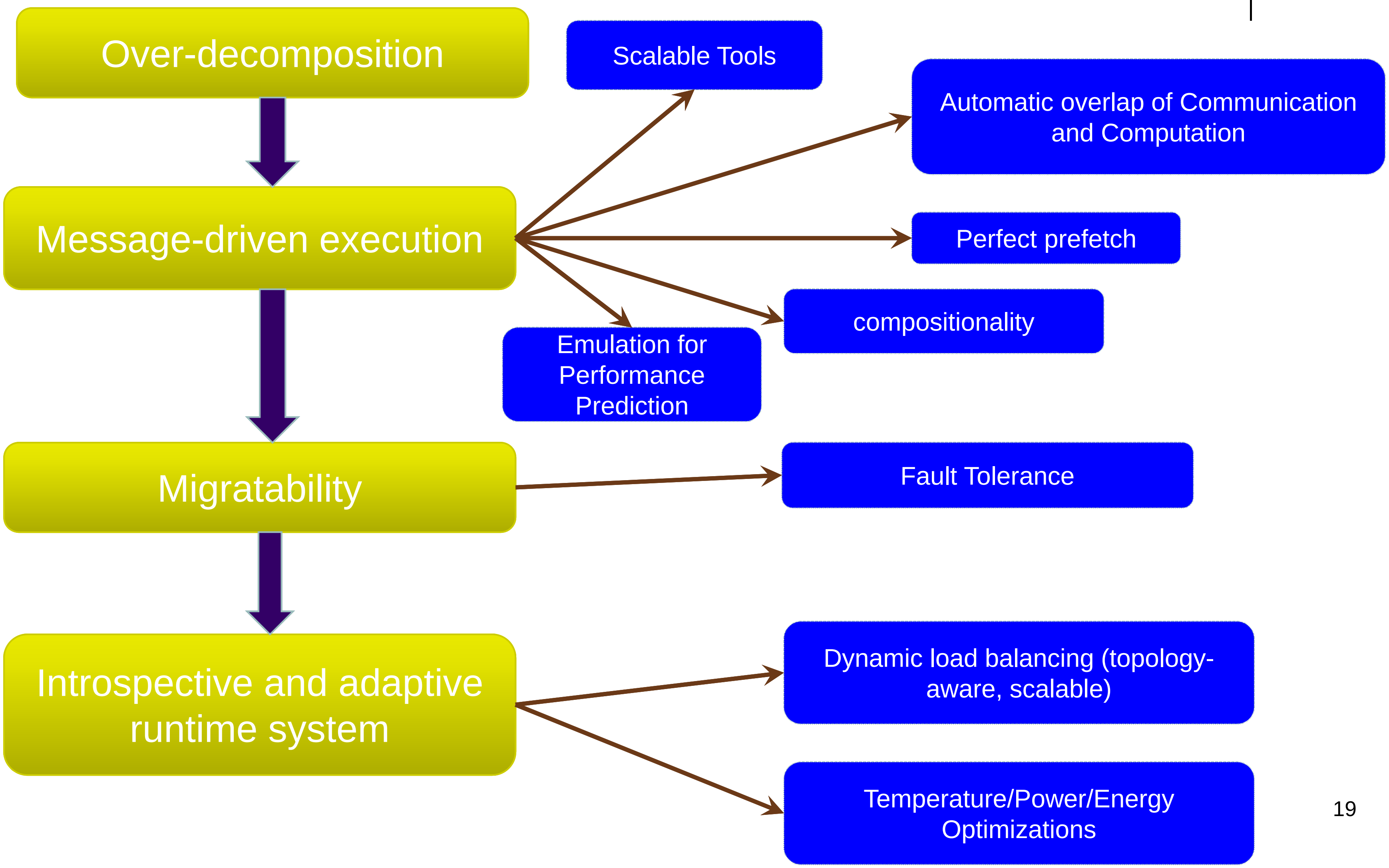

Processor 0
Scheduler
Message Queue
Processor 1
Scheduler
Message Queue
Processor 2
Scheduler
Message Queue
Processor 3
Scheduler
Message Queue

Empowering the RTS



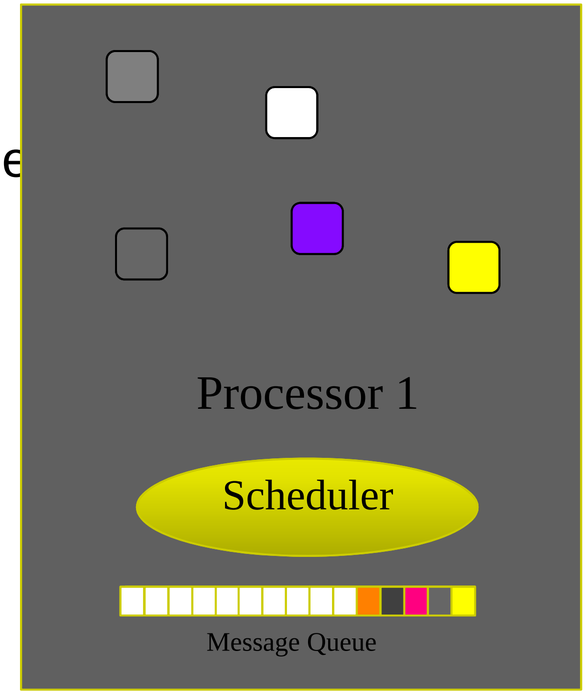
- The Adaptive RTS can:
  - Dynamically balance loads
  - Optimize communication:
    - Spread over time, async collectives
  - Automatic latency tolerance
  - Prefetch data with almost perfect predictability

Charm++ Position to Deliver at Many Ends
[see content of the blue boxes]



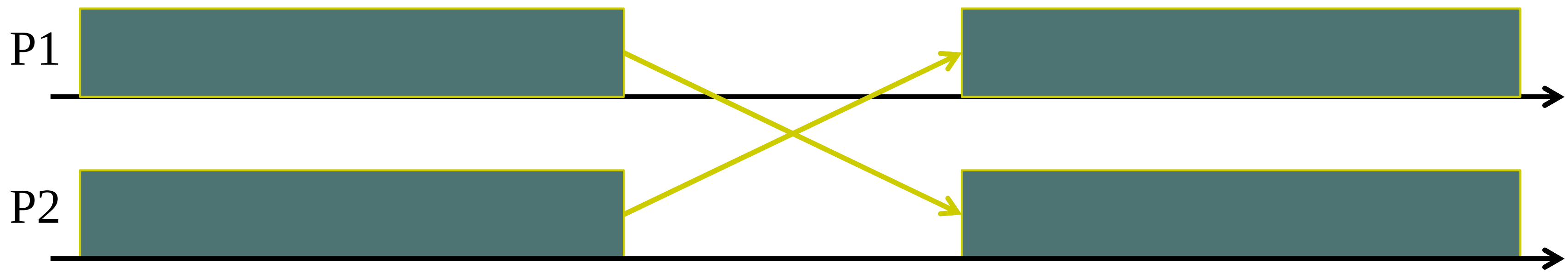
Utility for Multi-cores, Many-cores, Accelerators:

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



Impact on Communication

- Picture below: compute-communicate cycles in typical MPI apps

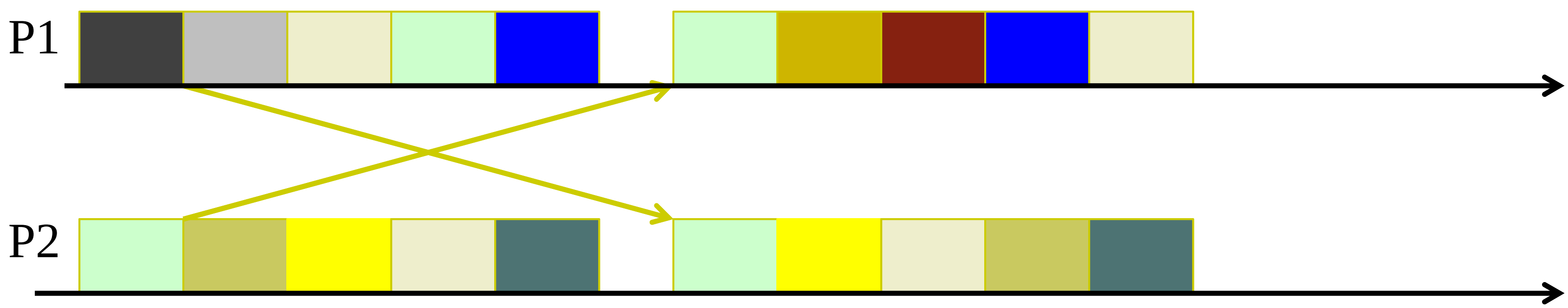


Bulk synchronous parallel (BSP) based application

- Use of communication network:
  - The network is used for a fraction of time
  - Turns out to be the bottleneck
- *Communication networks must be over-engineered for by necessity*

Impact on Communication

- With overdecomposition, as in Charm++
  - Communication is spread over an iteration
  - Facilitates adaptive overlap of communication and computation



Overdecomposition enables overlap

Decomposition Challenges

- Current method is to decompose to processors (MPI model)
  - Deciding which processor does what work in detail is difficult at large scale
  - Coordinating the message passing: why should the programmer be concerned with this?
- Charm++ take on this:
  - Decomposition should be independent of number of processors
  - The programmer should just figure our how to split the work/data into large count
- Adaptive scheduling of the objects on available resources done by the RTS

Decomposition in Charm++: Independent of Number of Cores

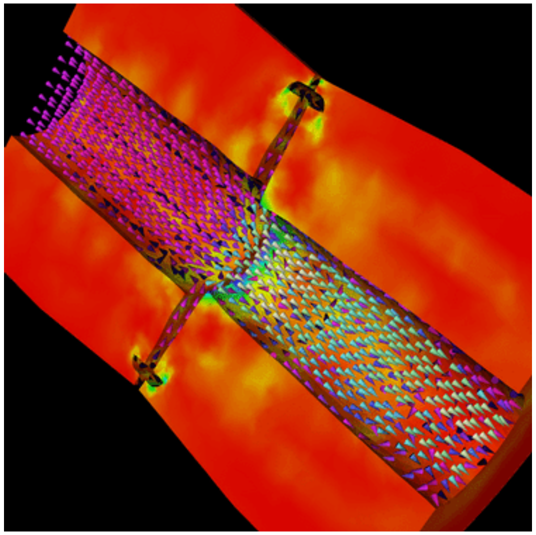
- Rocket simulation example under traditional MPI

1	2	. . .	P
Solid	Solid		Solid
Fluid	Fluid		Fluid

- With migratable-objects:

Solid$_1$ Solid$_2$ Solid$_3$ . . . Solid$_n$

Fluid$_1$ Fluid$_2$ . . . Fluid$_m$



Rocket Simulation UIUC

- Benefit: load balance, communication optimizations, modularity

So, What is Charm++?

- Charm++ is a way of parallel programming based on
  - Objects
  - Overdecomposition
  - Message
  - Asynchrony
  - Migratability
  - Runtime system

More Nuts and Bolts

Parallel Programming

- Decomposition
  - What to do in parallel
- Mapping:
  - Which PE does each task
- Scheduling (sequencing)
  - Done independently on each PE
- Machine dependent expression
  - Express the above decisions for the particular parallel machine

The “parallel objects model” of Charm++ automates the Mapping, Scheduling, and Machine dependent expression tasks

Charm++ Shared Objects Model

- Basic philosophy:
  - Let the programmer decide what to do in parallel
  - Let the system handle the rest:
    - Which PE executes what, and when
    - With some override control to the programmer, when needed
- Basic model:
  - The program is set of communicating objects
  - Objects only know about other objects (not processors)
  - System maps objects to processors
    - And may remap the objects for load balancing dynamically, at run time
- Shared objects, not shared memory

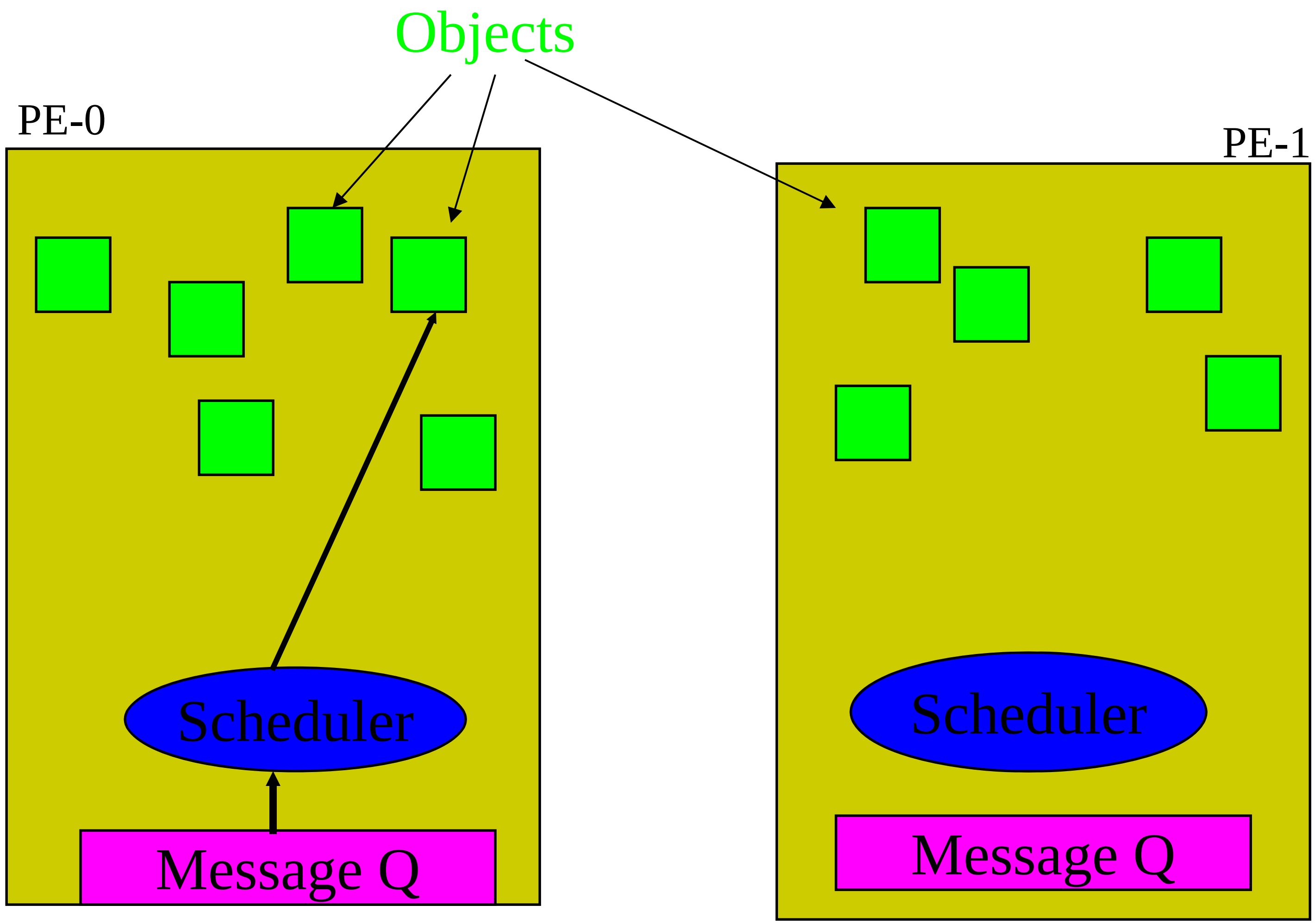
SAS – Shared Address Space

  - In-between “shared nothing” message passing, and “shared everything” of SAS
  - Additional information sharing mechanisms
  - “Disciplined” sharing

Charm++ Chares and Friends

- Charm++ programs specify parallel computations by relying on a number of “special objects”
  - Types of “special objects”
    - Chares: singleton objects
    - Chare arrays: generalized collections of objects
    - Chare group: advanced, used by library writers and the system
  - These objects communicate with each other
    - By invoking methods on each other
      - This is done *asynchronously* - key point
    - Also by sharing data using “specifically shared variables”

Charm++: Data Driven Execution



New Charm++ Concept: Proxy

- Consider the following scenario:
  - Object `x` of class `A` wants to invoke method `f` of object `y` of class `B`
  - `x` and `y` are managed by different PEs
  - What should the syntax be?
    - `y->f( …)` doesn't work because `y` is not a local pointer
- Solution in Charm++ uses a "proxy"
  - Instead of "`y`" we must use an ID that is valid across PEs
  - Rather than directly talking to `y`, method invocation should use this ID
  - Some part of the system must pack the parameters and send them
  - Some part of the system on the remote processor must invoke the right method on the right object with the parameters supplied

Charm++ Solution: Proxy Classes

- Classes with remotely invokeable methods
  - Inherit from “chare” class (system defined)
  - Entry methods can only have one parameter: a subclass of message
- If `D` is a chare class which has methods that we want to remotely invoke
  - The system will automatically generate a proxy class `Cproxy_D`
  - Proxy objects know where the real object is
  - Methods invoked on a proxy object simply put the data in an “envelope” and send it out to the destination
- Each chare object of type `D` has a proxy object

```
CProxy_D thisProxy; // thisProxy inherited from “CBase_D”
```

  - Also you can get a proxy for a chare when you create it:

```
CProxy_D myNewChare = CProxy_D::ckNew(arg);
```

Chare Creation and Method Invocation

```
CProxy_D x = CProxy_D::ckNew(25);
x.f(5,7);
```

Sequential equivalent:

```
y = new D(25);
y->f(5,7);
```

- Good to remember:
  - Each chare object of type `D` has a proxy object
  - A regular (non-chare) object does not have a proxy object

Chares: Syntax and Semantics

- A "chare": regular C++ class, with one caveat
  - It has some methods designated as remotely invokable
    - Called *entry methods* of that chare

Facility made available to you as soon as you indicate that a class (BB) is a chare class

- Chare creation (for chare class BB):

```
CProxy_BB  myChareProxy = CProxy_BB::ckNew(args);
```

  - Creates an instance of BB on a specified processor "pe"

```
CProxy_BB::ckNew (args, pe);
```

  - Cproxy_BB: a proxy class generated by Charm++ for chare class BB declared by you (the user)

Chares: Some Remarks

- You can regard a chare as a message (or data) driven object
  - The purpose of a chare is to be called
  - It is called by method invocation, like you'd call in C++
  - Since you don't know where the chare is, you access it via the proxy
  - In fact, a proxy is how a chare gets constructed
    - If you don't specify a PE, you essentially place a seed for a chare
    - In general, you don't specify a PE. Instead, you trust the RTS to choose the PE
      - RTS selects a PE to seed based on load balancing and other heuristics transparent to user

Chares: Some More Remarks

- The entry method definition specifies a function that is executed *without interruption* when a message is received and scheduled for processing.
- Only one message per chare is processed at a time.
- Order in which messages are process; i.e., methods are executed, is not deterministic
- Calls to entry methods are asynchronous
  - They must have the return type `void`

Proxy Class, Recap

- For each chare class BB, the system generates a proxy class
  - For `BB` we have `CProxy_BB` (no need to do anything, there for you)
- Global, in the sense of being valid on all processors
- thisProxy (analogous to this) gets you your own proxy
- You can send proxies in messages
- Given a proxy `p`, you can invoke methods:

```
p.method(msg);
```

- Conclusion: chares live on some PE, yet a proxy to it can be sent to, and used by any other object argc/argv

Execution begins here

```
BB::BB(CkArgMsg * m)
{
    responders    = 100;  // member of BB, might be set via m
    numberOfDraws = 1000; // member of BB, might be set via m
    count = 0;
    const int participants = responders;

    for (int i = 0; i< participants; i++)
        new CProxy_piPart(thisProxy, numberOfDraws);
}

void BB::results(int pcount)
{
    count += pcount;
    if (--responders == 0) {
        cout << "pi=: " << 4.0*count/(participants*numberOfDraws) << endl;
        CkExit();
    }
}
```

Exit the program (called once)

```
piPart::piPart( CProxy_BB& BBProxy, int draws )
{
    // declarations here…
    drand48((long) this);

    for (i = 0; i < draws; i++) {
        x = drand48();
        y = drand48();
        if ((x*x + y*y) <= 1.0) localCount++;
    }
    BBProxy.results(localCount);
    delete this;
}
```

Generation of Proxy Classes

[talking on this slide about classes, not objects!]

- How does Charm++ generate the proxy classes?
  - Needs help from the programmer, at run time
    - You have to indicate the classes & methods that can be remotely invoked
      - Declared in a special “charm interface” file (see `pgm.ci` below)
    - You have to include the generated code in your program
      - What gets generated: `PiMod.decl.h` and `PiMod.def.h`

pgm.ci

```
mainmodule PiMod {
mainchare BB {
   entry BB(CkArgMsg * m);
   entry results(int pc);
};
chare piPart {
   entry piPart(CProxy_BB&, int);
};
```

Compile driver generates two files:

`PiMod.decl.h`

`PiMod.def.h`

The `pgm.h` file

```
#include “PiMod.decl.h”

..
```

The `Pgm.c` file

```
...
#include “PiMod.def.h”
```

Another Example: With and Without Communication

- Hello World! example, done one of two ways:
  - No chare invocation
  - W/ chare invocation
- Touches on: the Charm++ build/run process

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”
```

Hello World!, with Chares

`hello.ci` file

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

  chare WhatsUp {
    entry WhatsUp();
  };
};
```

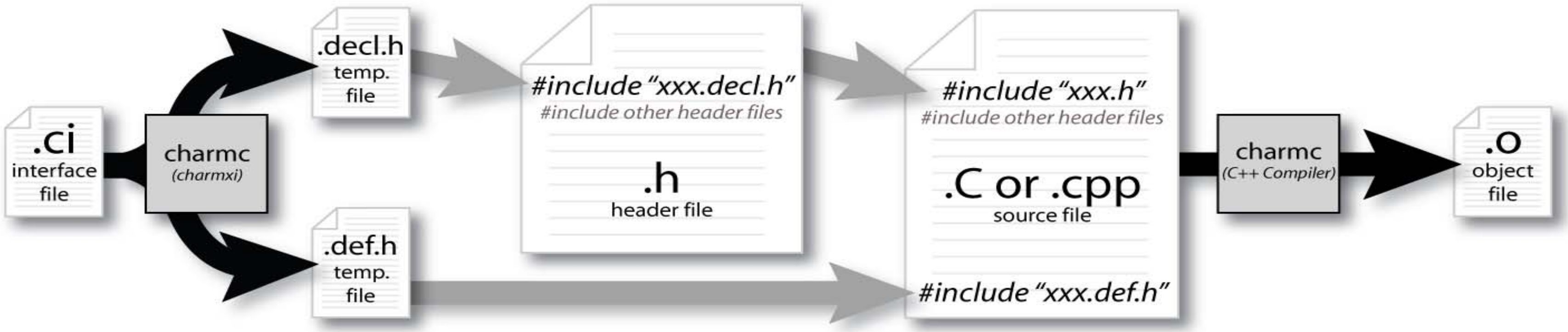
`hello.cpp` file

```
#include <stdio.h>
#include “hello.decl.h”

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

class WhatsUp :
          public CBase_WhatsUp {
  public: WhatsUp() {
    ckout<<“Hello World!”<<endl;
    CkExit();
  };
};
#include ”hello.def.h”
```

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 typically creates some additional chares

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

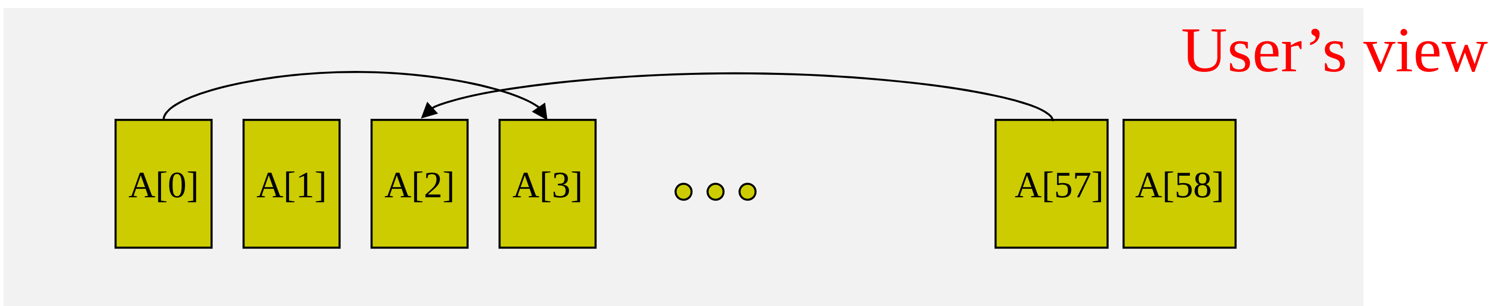
```
>> ./charmrun +p7 ./hello
```

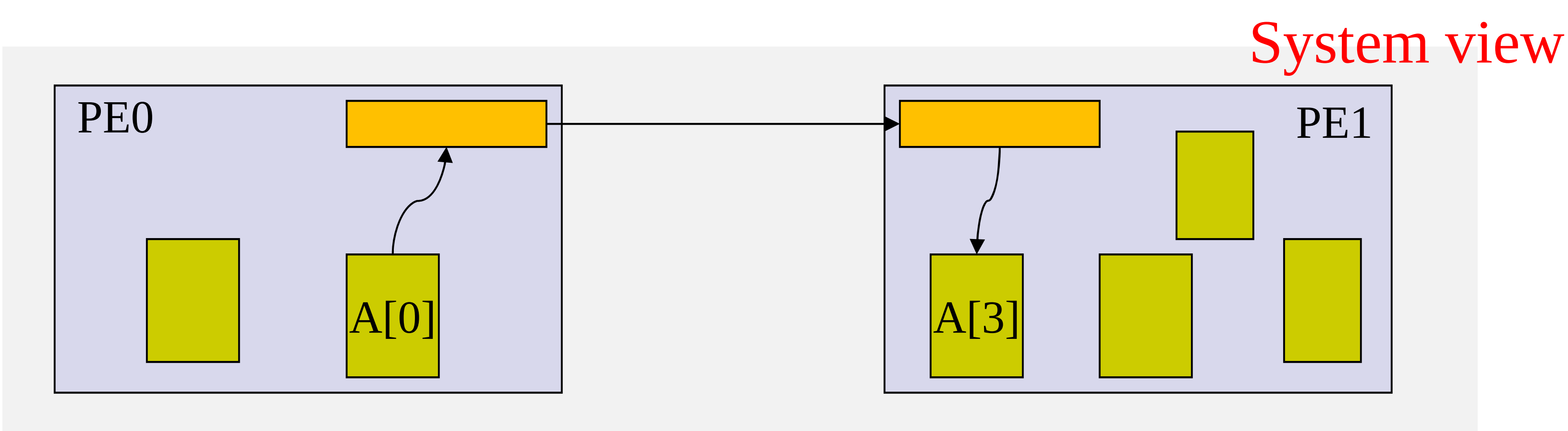
(the +p7 tells the system to use seven cores)

[New Topic]

Chare Arrays

- Each chare member addressed by an index
- Mapping of element objects to PEs handled by the RTS





Chare Arrays

- Elements are indexed by a user-defined data type
  - [sparse] 1D, 2D, 3D, tree, ...
  - Index used to identify which particular object you interact with
- Reductions and broadcasts across the array
- RTS supports dynamic insertion, deletion, migration
- Arrays coordinate with automatic load balancer
  - Very nice feature…

1D Declare & Use

```
module m {
  array [1D] Hello {
    entry Hello(void);
    entry void SayHi(int HiData);
  };
};
```

In the interface (.ci) file

In the .C file

```
//Create an array of Hello's with 4 elements:
const int nElements=4;
CProxy_Hello p = CProxy_Hello::ckNew(nElements);
//Have element 2 say "hi"
P[2].SayHi(991);
```

1D Definition

```
class Hello: public CBase_Hello {
public:
  Hello(void) {
    … thisProxy …
    … thisIndex …
  }
  void SayHi(int m) {
     if (m <1000)
          thisProxy[thisIndex+1].SayHi(m+1);
  }

  Hello(CkMigrateMessage *m) {}
};
```

Inherited from ArrayElement1D

Collective Ops on an Array "p"

- Array level execution of `SayHi`:
  `p.SayHi(data);`

- Reduce x across all elements:
  `contribute(sizeof(x), &x, CkReduction::sum_int, cb);`

- Where do reduction results go?
  To a "callback" function, named *cb* above:

```
// Call some function foo with fooData when done:
CkCallback cb(foo,fooData);

// Broadcast the results to my method "bar" when done:
CkCallback cb(CkIndex_MyArray::bar,thisProxy);
```

Charm++ Migration Support

- Delete element i:

```
p[i].destroy();
```

- Migrate to processor `destPe`:

```
migrateMe(destPe);
```

- Load balancer: there is a native Charm++ one
  - User can create his/her own
  - Essential component: pack/unpack (pup) function
  - Each migratable object provides a “pup” method
    - pup is a single abstraction that allows data traversal for determining size, packing and unpacking

Information Sharing Abstractions

- Observation:
  - Information is shared in several specific modes in parallel programs

- Other models support only a limited sets of modes:
  - Shared memory: everything is shared $\rightarrow$ sledgehammer approach
  - Message passing: messages are the only method

- Charm++: identifies and supports several modes
  - readonly / writeonce
  - tables (hash tables)
  - accumulators
  - monotonic variables

Charm++ Concerns
[personal, maybe they're nonissues]

- Large codes handling lots of chares $\rightarrow$ proxies must induce some overhead
  - They must exist at global scope. Is this a bottleneck?
- Surprisingly little software developed using Charm++
  - One good example: NAMD2
- Memory consistency model: I need to look into this
- SDAG: how general is this mechanism? What is the interplay between SDAG and memory consistency model in Charm++
  - Probably spelled out somewhere, should spend more time thinking about this
- Lack of libraries that one case use
  - Solution of sparse linear systems, PDE solvers, optimization, etc.
    - You have start with a clean slate

ME759 Wrap-up

ME759 Parallelism We Touched Upon

[to a larger or smaller extent]

Cluster	Have discussed in some detail	Group of computers communicating through fast interconnect
Coprocessors/Accelerators	Have discussed in some detail	Special compute devices attached to the local node through special interconnect
Node	Have discussed in some detail	Group of processors communicating through shared memory
Socket	Have discussed in some detail	Group of cores communicating through shared cache
Core	Have discussed in some detail	Group of functional units communicating through registers
Hyper-Threads	Have discussed in some detail	Group of thread contexts sharing functional units
Superscalar	Have discussed, but little direct control	Group of instructions sharing functional units
Pipeline	Have discussed, but little direct control	Sequence of instructions sharing functional units
Vector	Have discussed in some detail	Single instruction using multiple functional units

Have discussed in some detail → [green]

Have discussed, but little direct control → [yellow]

[Intel]→

Skills I hope You Picked Up in ME759

- I think of these as items that you can add to your resume:
  - Basic understanding of what happens on one core
  - Basic understanding of hardware for parallel computing
  - CUDA programming
  - OpenMP Programming
  - MPI Programming
  - Understanding of the parallel computing model induced by each solution; i.e., MPI, OpenMP, or CUDA
  - Basic understanding of the parallel computing landscape
    - From AVX vectorization to parallel computing w/ Charm++

ME759: Most Important Three Things

- Hone your “computational thinking” skills
- Don’t move data around
  - Costly in terms of time and energy
- Expose concurrency/parallelism in your solution