Charm++ update

Laxkinant “Sanjay” Kale
Parallel Programming Laboratory
Univ of Illinois

PPL
UIUC

Our Guiding Principles: in Charm

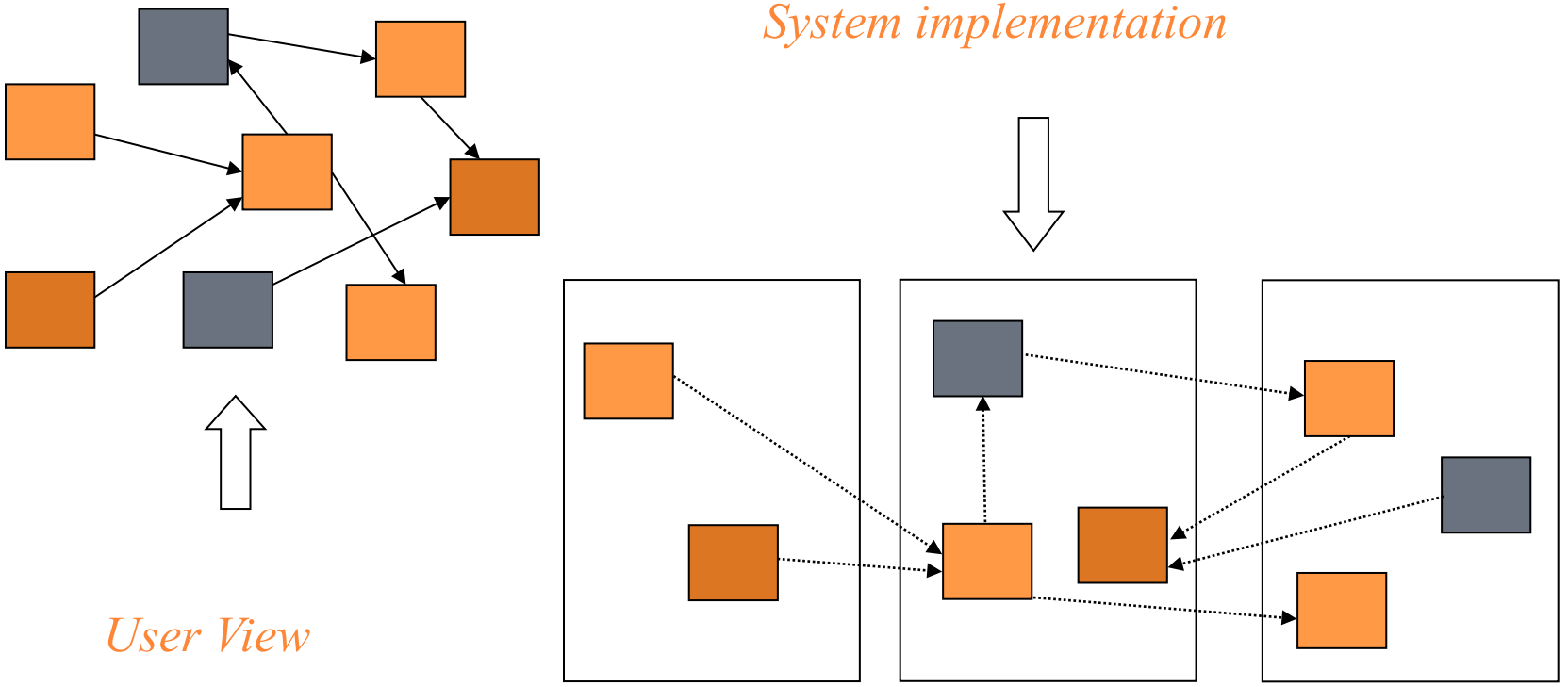
- No reliance on magic
  - (at least until you learn the trick)
  - Parallelizing compilers have achieved close to technical perfection, but are not enough
  - Sequential programs obscure too much information
- Seek an optimal division of labor between the system and the programmer
- Design abstractions based solidly on use-cases
  - Application-oriented yet computer-science centered approach

Object based over-decomposition

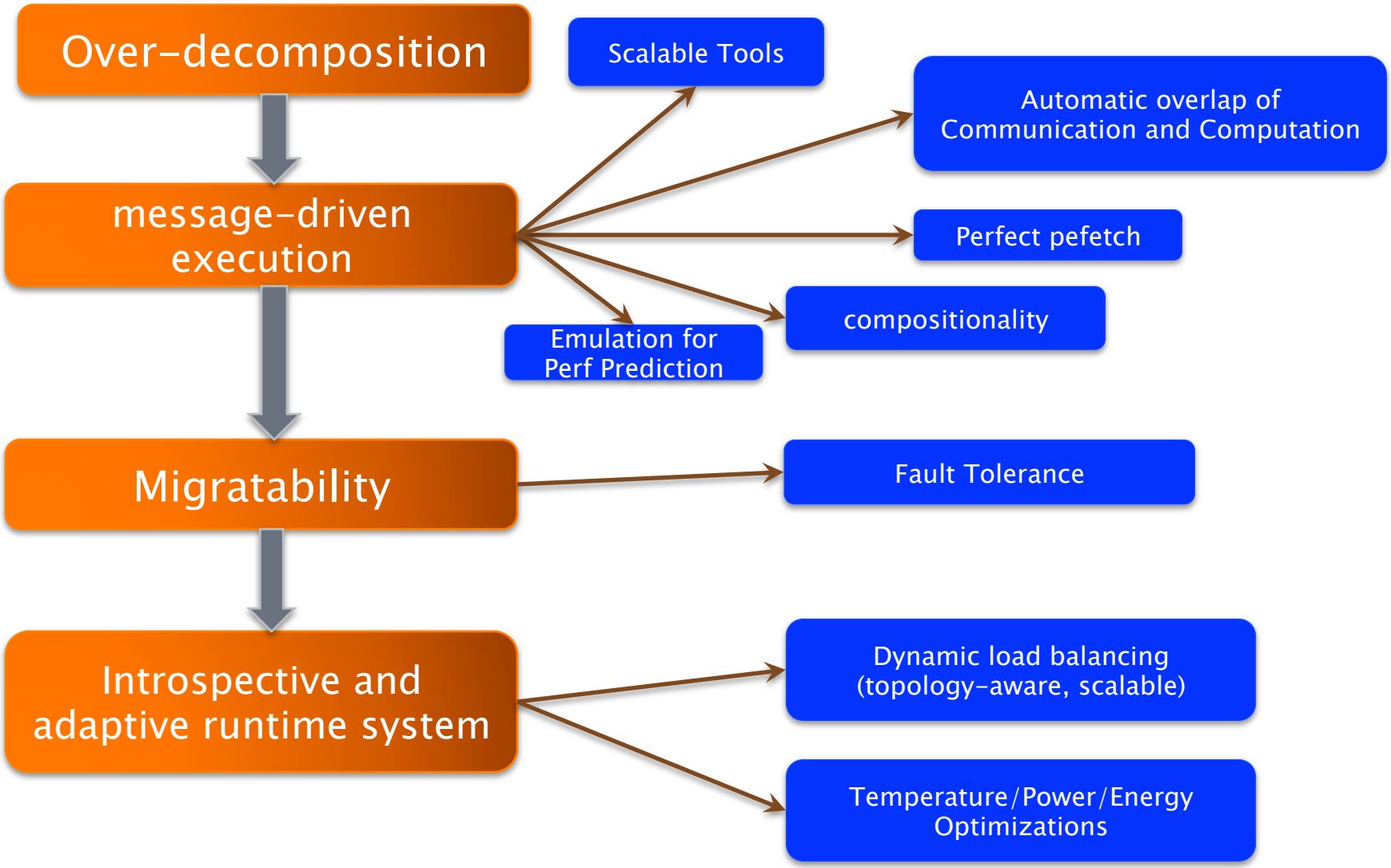
- Let the programmer decompose computation into objects
  - Work units, data-units, composites
- Let an intelligent runtime system assign objects to processors
  - RTS can change this assignment during execution
- This empowers the RTS
  - The research agenda started with the simple precept above, just before NAMD,
  - Continued until now!

Object-based over-decomposition: Charm++

- Multiple "indexed collections" of C++ objects
- Indices can be multi-dimensional and/or sparse
- Programmer expresses communication between objects
  - with no reference to processors

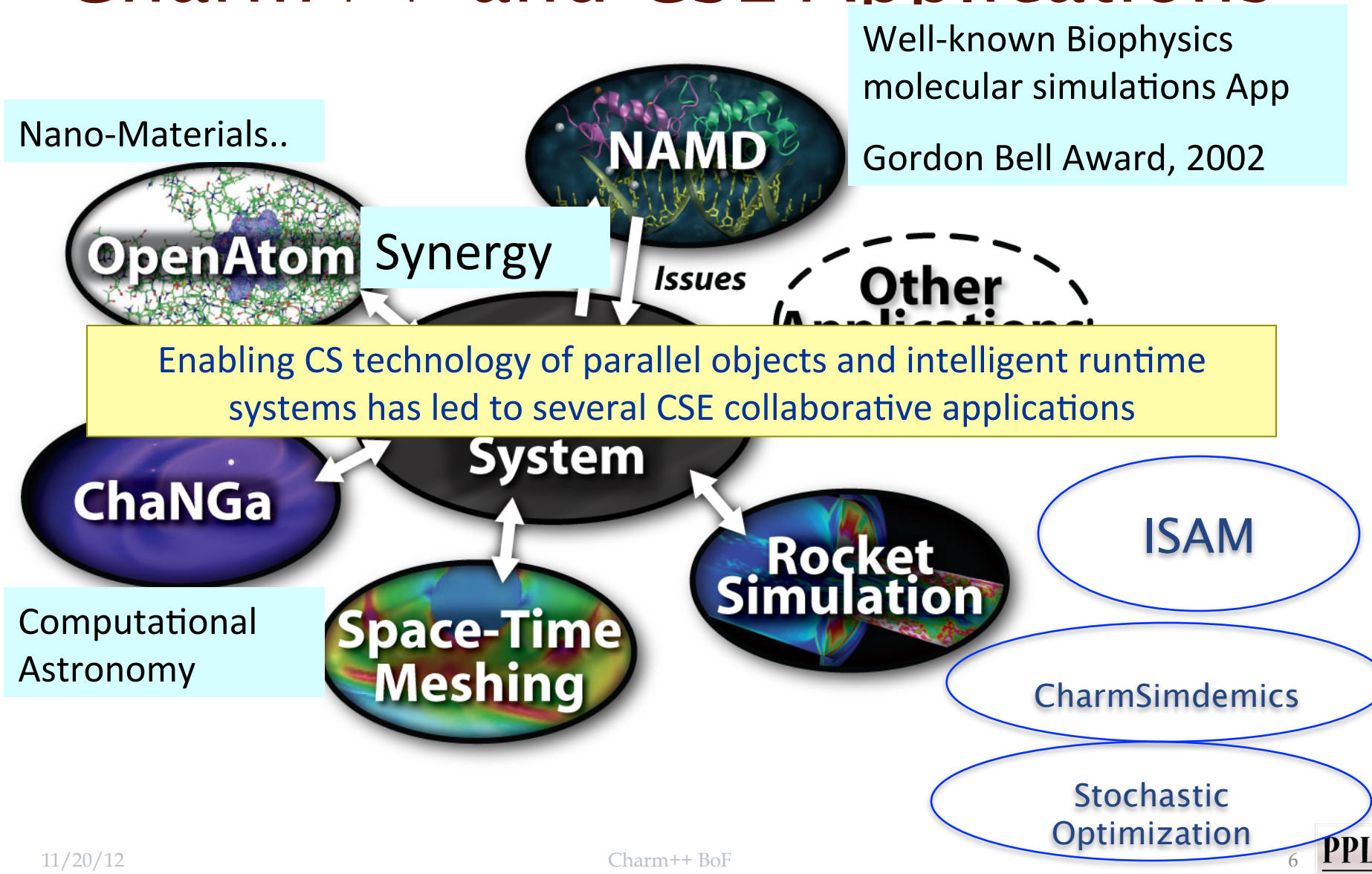


Benefits of Charm++ model



PPL UIUC

Charm++ and CSE Applications

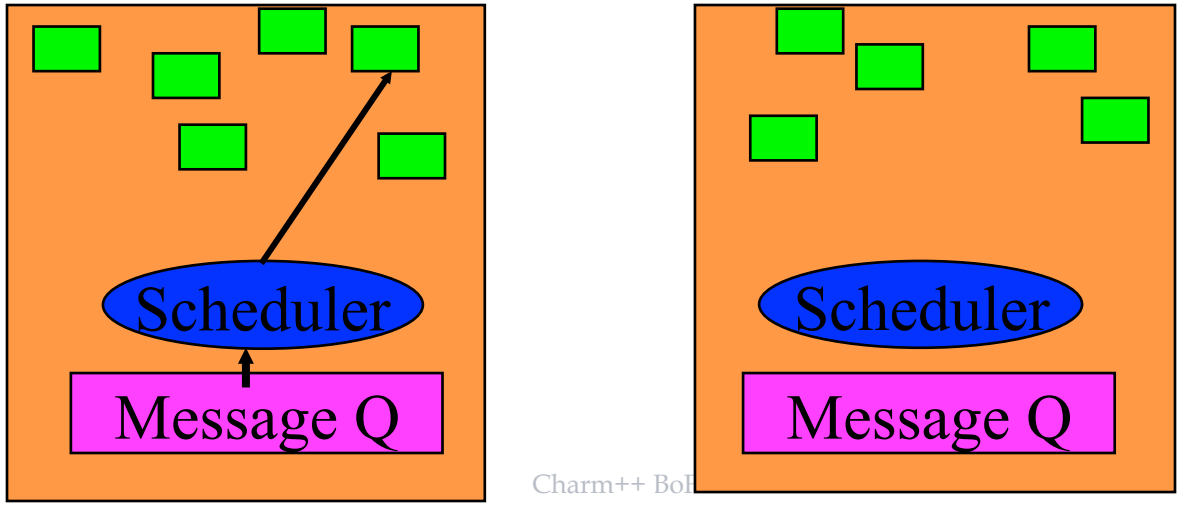


Adaptive Runtime Systems

- Decomposing program into a large number of WUDUs empowers the RTS, which can:
  - Migrate WUDUs at will
  - Schedule DEBS at will
  - Instrument computation and communication at the level of these logical units
    - WUDU x communicates y bytes to WUDU z every iteration
    - SEB A has a high cache miss ratio
  - Maintain historical data to track changes in application behavior
    - Historical => previous iterations
    - E.g., to trigger load balancing

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

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



Impact on communication

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

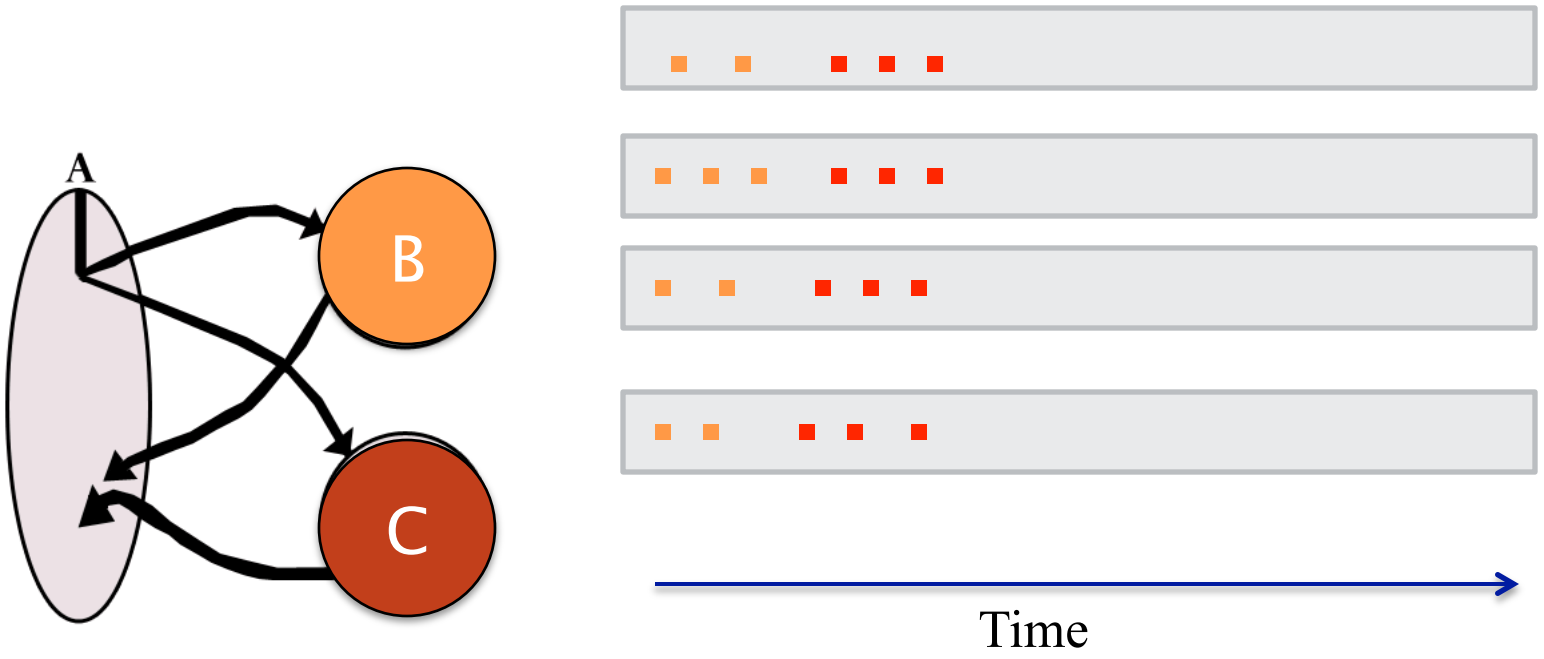
Compositionality

- It is important to support parallel composition
  - For multi-module, multi-physics, multi-paradigm applications…
- What I mean by parallel composition
  - B || C where B, C are independently developed modules
  - B is parallel module by itself, and so is C
  - Programmers who wrote B were unaware of C
  - No dependency between B and C
- This is not supported well by MPI
  - Developers support it by breaking abstraction boundaries
    - E.g., wildcard recvs in module A to process messages for module B
  - Nor by OpenMP implementations:

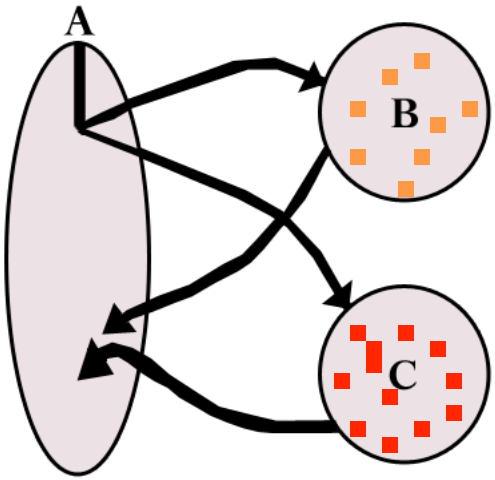
Without message-driven execution (and virtualization), you get either:

Space-division

OR: Sequentialization



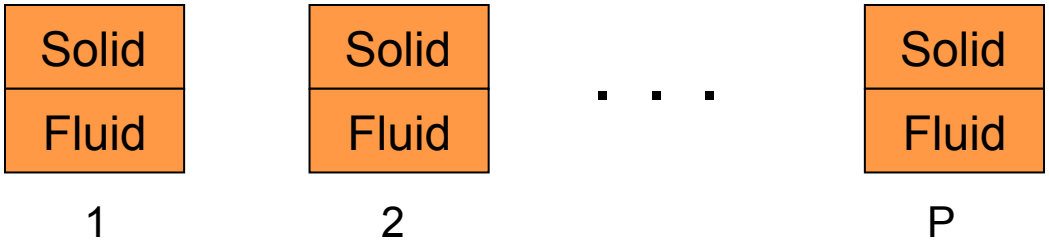
Parallel Composition: A1; (B || C); A2

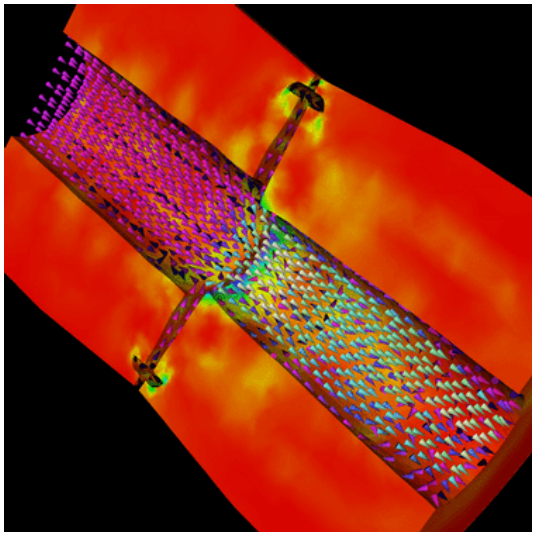


Recall: Different modules, written in different languages/paradigms, can overlap in time and on processors, without programmer having to worry about this explicitly

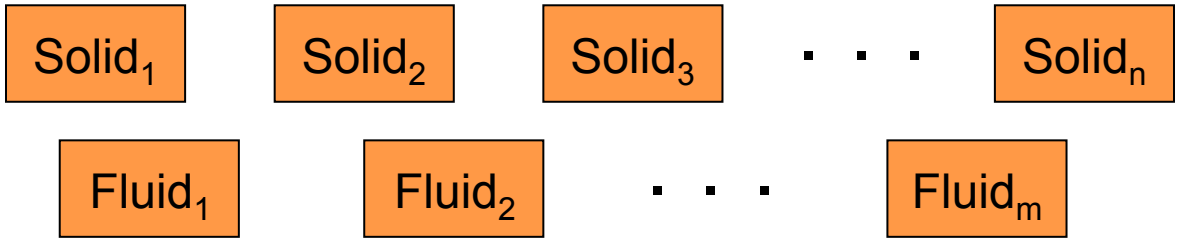
Decomposition Independent of numCores

- Rocket simulation example under traditional MPI





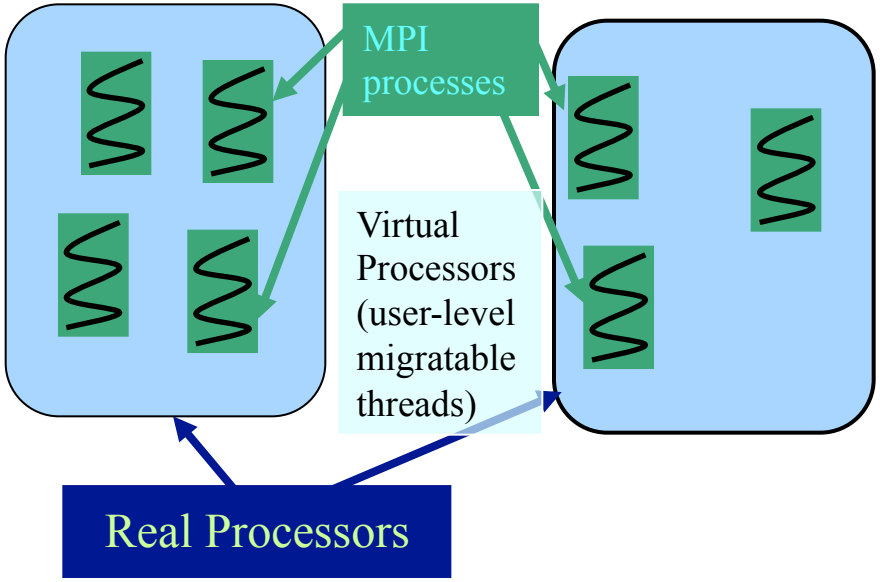
- With migratable-objects:



  - Benefit: load balance, communication optimizations, modularity

Object Based Over-decomposition: AMPI

- Each MPI process is implemented as a user-level thread
- Threads are light-weight and migratable!
  - <1 microsecond context switch time, potentially >100k threads per core
- Each thread is embedded in a charm++ object (chare)



Saving Cooling Energy

- Easy: increase A/C setting
  - But: some cores may get too hot
- Reduce frequency if temperature is high
  - Independently for each core or chip
- This creates a load imbalance!
- Migrate objects away from the slowed-down Procs
  - Balance load using an existing strategy
  - Strategies take speed of processors into account
- Implemented in experimental version
  - SC 2011 paper
  - IEEE TC paper
- Several new power/energy-related strategies
  - PASA '12: Exploiting differential sensitivities of diff code segments to freq change

Fault Tolerance in Charm++/AMPI

- Four Approaches:
  - Disk-based checkpoint/restart
  - In-memory double checkpoint/restart
  - Proactive object migration
  - Message-logging: scalable fault tolerance
- Common Features:
  - Leverages object-migration capabilities
  - Based on dynamic runtime capabilities
- Several new results in the last year:
  - FTXS 2012: scalability of in-mem scheme
  - Hiding checkpoint overhead .. with semi-blocking..
  - Energy efficiency of FT protocols : best paper SBAC-PAD

HPC Challenge Class 2 Award

- Class: about programming systems
- 2011: Charm++ won the award with Chapel
- 2012: Charm++ was a finalist
- http://charm.cs.illinois.edu/papers/12-47

Our HPCC submission: summary

Productivity						Performance		
Code	C++	CI	Benchmark Subtotal	Driver	Total	Machine	Max Cores	Performance Highlight
Required Benchmarks								
1D FFT	54	29	83	102	185	BG/P BG/Q	64K 16K	2.71 TFlop/s 2.31 TFlop/s
Random Access	76	15	91	47	138	BG/P BG/Q	128K 16K	43.10 GUPS 15.00 GUPS
Dense LU	1001	316	1317	453	1770	XT5	8K	55.1 TFlop/s (65.7% peak)
Additional Benchmarks								
Molecular Dynamics	571	122	693	n/a	693	BG/P BG/Q	128K 16K	24 ms/step (2.8M atoms) 44 ms/step (2.8M atoms)
AMR	1126	118	1244	n/a	1244	BG/Q	32k	22 steps/sec, 2d mesh, max 15 levels refinement
Triangular Solver	642	50	692	56	748	BG/P	512	48x speedup on 64 cores with helm2d03 matrix

http://charm.cs.illinois.edu/papers/12-47

Other papers and work

- uGNI port and optimizations for NAMD: SC12
- PAMI port
- Improved support for sections (unranked)
- Meta-Balancer
- AMR: Highly asynchronous and memory efficient
- Charm++ for multicore systems: Best paper HiPC'11
- Charm in the cloud:
- Next : Charj

Charj: Compiler Supported Language with an Adaptive Runtime

Laxmikant (Sanjay) Kale,
http://charm.cs.illinois.edu

Based on Aaron Becker's thesis work

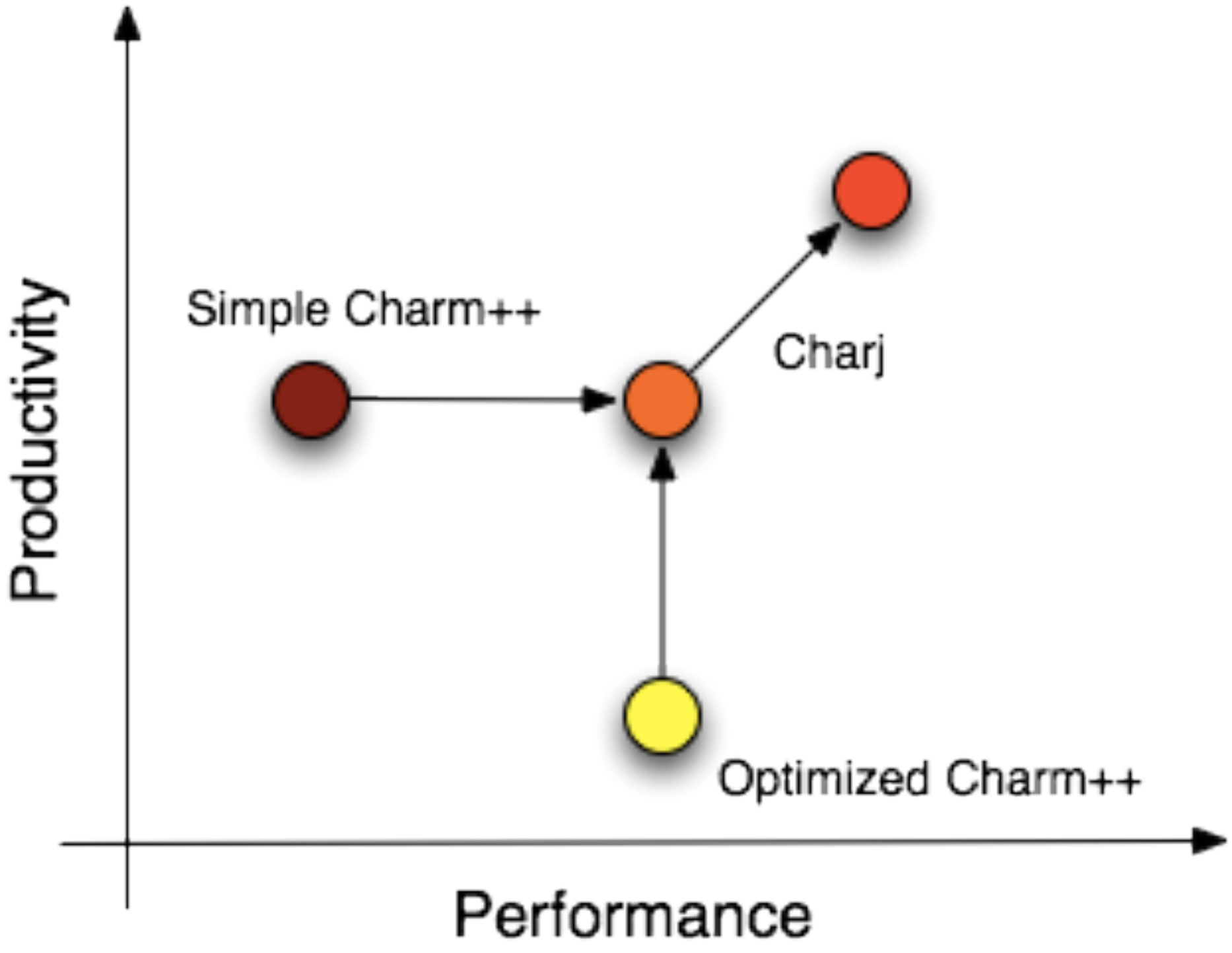
Thesis

Simple compiler support and basic static analysis can, when paired with a sophisticated and feature-rich runtime system, significantly improve productivity.

Approach

- Combine compiler technology with a rich runtime system to increase productivity without harming performance
  - Better safety checks by incorporating programming model semantic knowledge into compiler
  - Static analysis allows more enforcement and can provide optimizations that are impossible at the library level
  - Tightly integrate with multiple programming models
  - Add language-level support for rich runtime features

Where can I find a rich adaptive RTS?


Productivity
Simple Charm++
Charj
Optimized Charm++
Performance

The Charj Language

Compiler Infrastructure

- Multi-pass compiler written in Java, uses ANTLR compiler construction tool
- Simple operations use ANTLR's AST recognition and rewriting features
- More complex operations operate directly on the AST and construct an explicit CFG
- Supports inter-procedural data-flow analysis
- Compiler driver takes .cj input, produces C++ and .ci files, and translates and compiles the output source

Problems with Charm

- Most of your code is only seen by a C++ compiler
- No way to do lots of simple things, especially:
- Enforce Charm semantics
- Do compile-time analysis and optimization
- Moving model-specific features into the interface file works, but it's difficult and inflexible.

Charj Design Principles

- Keep it simple
- Minimize new syntax
- Distinguish between local and remote operations
- Integrate tightly with the runtime

Productivity Benefits

- Enforcement of programming model semantics by the compiler (e.g. assignment of readonly variables)
- Elimination of redundant program information
- Improved messages for Charm-specific syntax errors
- Clear syntactic distinction between remote and local operations
- Optimizations can be done by compiler instead of by hand

Example: Readonly Variables

```
int n; // readonly variable
...
n = 17; // Ok if we're in a
        // mainchare constructor.
        // Silent bug otherwise.
```

Example: Readonly Variables

In Charj:

```
readonly int n;
...
n = 17; // Compiler will notify
        // the programmer of
        // illegal assignments
```

Example: Custom Reducers

```
CkReductionMsg* _my_reducer(
        int nMsg, CkReductionMsg** msgs)
{
    MyType* accum = new MyType();
    for (int i=0; i<nMsg; ++i) {
        MyType* x;
        PUP::fromMem p(msgs[i]->getData());
        p | *x;
        accum->reduce(x);
    }
    return CkReductionMsg::buildNew(...);
}
```

Example: Custom Reducers

```
// .ci
initcall void _register_my_reducer(void);

// .cc
CkReduction::reducerType _my_reducer_type;
void _register_my_reducer(void)
{
    _my_reducer_type =
      CkReduction::addReducer(_my_reducer);
}
```

Example: Custom Reducers in Charj

```
reducer<MyType> my_reducer {
    my_reducer() { accum = new MyType(); }
    reduce(MyType x) { accum.reduce(x); }
}
```

Embedded Programming Models

Structured Dagger

- Coordination mini-language implemented on the Charm runtime
- Implemented as library + translator, functions containing SDAG are put into Charm interface files, translator emits C++
- Allows the programmer to express the parallel structure of an object's lifetime without the need for threading or blocking constructs
- Allows clear, concise, efficient code

Stencil code with sdag

```
entry void stencil()
{
    for (int i=0; i<N; ++i) {
        sendStrips();
        overlap {
            when getStripFromLeft(Strip s) {
                processStripFromLeft(s);
            }
            when getStripFromRight(Strip s) {
                processStripFromRight(s);
            }
        }
        doStencil();
    }
}
```

Stencil: Message Driven Equivalent

```
entry void stencil()
{
    i = 0;
    mainLoop();
}

void mainLoop()
{
    leftStripReceived = false;
    rightStripReceived = false;
    if (i < N) {
        sendStrips();
    }
}
```

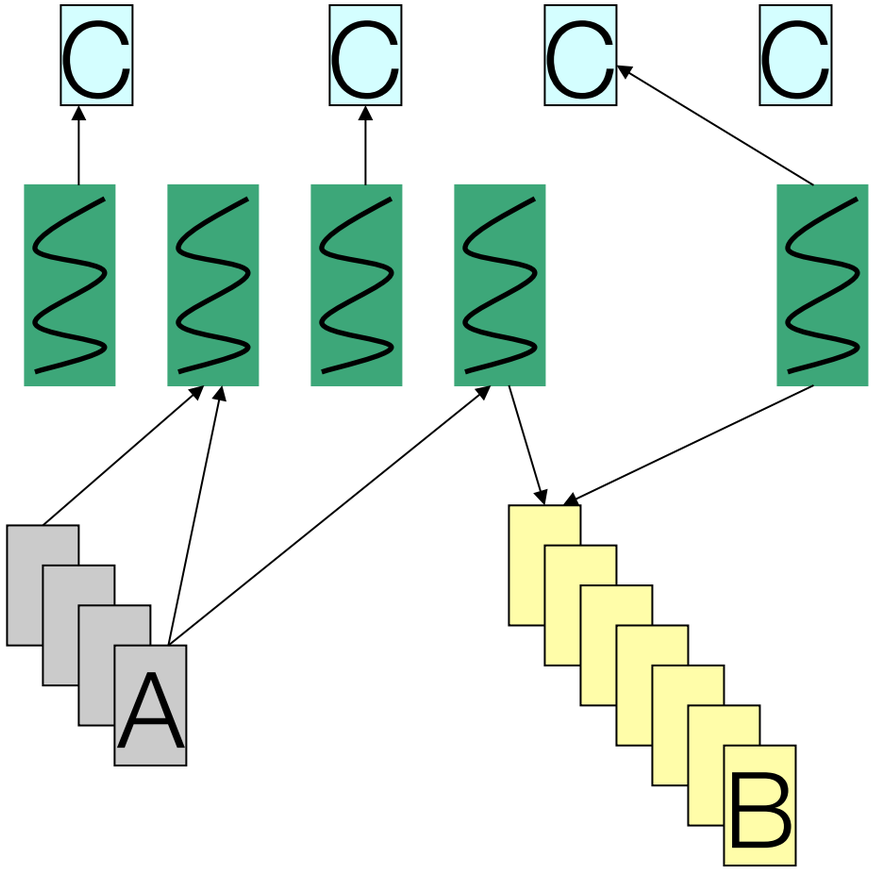
```
entry void getStripFromLeft(Strip s)
{
    processStripFromLeft(s);
    leftStripReceived = true;
    checkOverlapCompletion();
}

entry void getStripFromRight(Strip s)
{
    processStripFromRight(s);
    rightStripReceived = true;
    checkOverlapCompletion();
}

void checkOverlapCompletion()
{
    if (leftStripReceived && rightStripReceived) {
        doStencil();
        ++i;
        mainLoop();
    }
}
```

Improvements in Charj SDAG

- Local variables
- Free mixing of sequential constructs and SDAG constructs (no “atomic” blocks)
- No redundant declarations for “when” triggers
- No need for macro insertion or initialization calls during construction and migration

Multiphase Shared Arrays

- Disciplined access to arrays in a partitioned global address space
- Arrays go through *phases*, with synchronization between
- In each phase, only a subset of accesses are legal (e.g. read-only, write-only, accumulate)



MSA: Original Implementation

```
// MSA array A in write mode
for (int i=0; i<N; ++i)
    A[random()]++;

A.sync(); // transition to read mode

for (int i=0; i<N; ++i)
    printf(“%d ”, A[i];
```

MSA: Typed Handles

```
// MSA array A in write mode
MSA::Write whandle = A.getInitialWrite();
for (int i=0; i<N; ++i)
    whandle(random())++;

MSA::Read rhandle = whandle.syncToRead();

for (int i=0; i<N; ++i)
   printf("%d ", rhandle(i));
```

MSA: Charj Implementation

```
// MSA array A in accumulate mode
for (int i=0; i<N; ++i)
    A[random()]++;

A.syncToRead();

for (int i=0; i<N; ++i)
    printf(“%d ”, A[i]);
```

Accelerated Entry Methods

- Access to different types of accelerator hardware using a unified programming model and syntax
- Programmer creates special *accelerated* entry methods using a variant of normal entry method syntax
- Entry method is split into two pieces: body (can execute on host or on accelerator) and callback (host only)
- Runtime system can execute them on either the host processor or on accelerator hardware

Accelerated Entry Methods

```
entry [accel] void X(int n)
[ readWrite : float A <impl_obj->A>,
  readOnly : float B <impl_obj->B> ]
{
   // ...
} x_callback;
```

Accelerated Entry Methods

```
entry [accel] void X(int n)
[ readWrite : float A <impl_obj->A>,
  readOnly : float B <impl_obj->B> ]
{
   // ...
} x_callback;
```

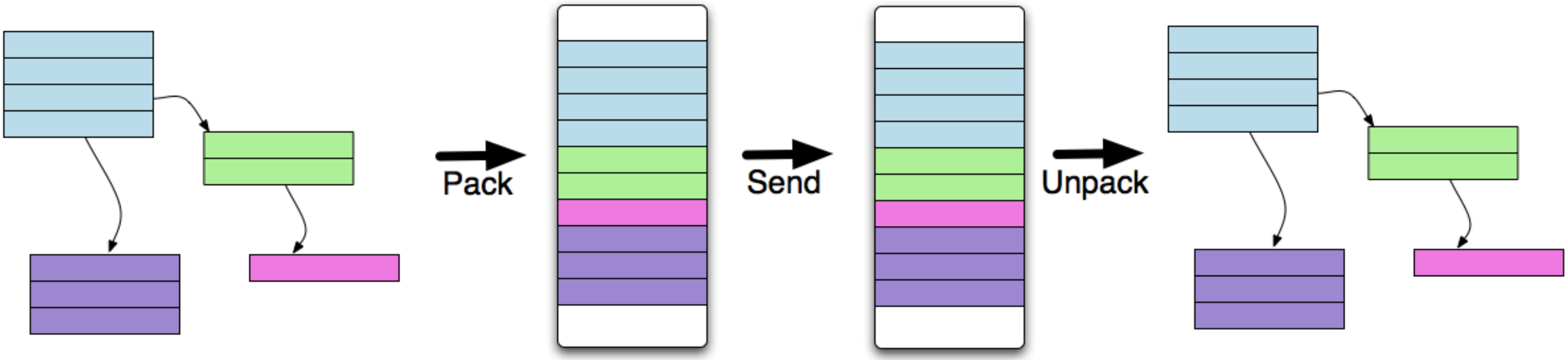
Accelerated Entry Methods

```
accelerated entry void X(int n) {
   // ...
} x_callback;
```

Optimizations

Packing and Unpacking

How do we communicate data structures in a parallel application?



MSA Strip Mining

- MSAs are split into *pages*, and MSA accesses go through a local page cache
- Generic array accesses must first check to see if the desired element is locally available, and if not, fetch it
- Prefetching and raw array accesses are faster, but more work for the programmer

MSA Strip Mining

```
for (int i=0; i<N; ++i)
  x = f(A[i]);
```

MSA Strip Mining

```
fetchPage(A, 0);
for (int i=0; i<N/PAGE; ++i) {
    if (i+1 < N/PAGE)
        fetchPage(A, i+1);
    waitForPage(A, i);
    for (int j=i*PAGE; j<(i+1)*PAGE; ++j)
        x = f(A.rawAccess(j));
}
```

Charj Application Suite

- LU Decomposition
- LeanMD (Molecular Dynamics)
- Barnes-Hut
- Jacobi Relaxation

Charj Application Suite

Source Lines of Code

	Charm	Charj	% Reduction
LU	187	135	28%
LeanMD	941	683	27%
Barnes-Hut	5174	3808	26%
Jacobi	327	163	50%

Contributions

- Demonstration of the thesis via the development of Charj programs that are simpler than their Charm equivalents
- A language targeting the Charm runtime system that supports multiple embedded programming models.
- A compiler for that language, supporting semantic checks and optimizations specific to Charj.
- Embeddings of multiple DSLs based on the Charm runtime into Charj
- A collection of Charj implementations of existing applications, which demonstrate the features of Charj.

Summary

- By combining compiler techniques with a rich runtime system, we can improve programmer productivity without sacrificing performance
    - Improved syntax and semantic checks
    - Better integration of multiple programming models
    - Optimizations powered by static analysis

More Info: http://charm.cs.illinois.edu/