Compiler Support for Productive Message-Driven Parallel Programming

Aaron Becker, PPL/UIUC

Final Defense
5 June 2012

Productivity in High Performance Computing

- Creating fast, scalable parallel applications is hard
- Productivity for HPC programmers is notoriously poor
- As machines get larger, the problem only gets worse

Productivity in High Performance Computing

- Creating fast, scalable parallel applications is hard
- Productivity for HPC programmers is notoriously poor
- As machines get larger, the problem only gets worse

- But, people are writing bigger, more scalable applications than ever

Productivity in High Performance Computing

- What have people been using?
  - Low-level, proven systems (MPI, GA, Pthreads)
  - New from-scratch parallel languages (X10, Chapel)
  - Narrowly targeted compiler support (CAF, TCE)
  - Auto-parallelizing compilers (???)

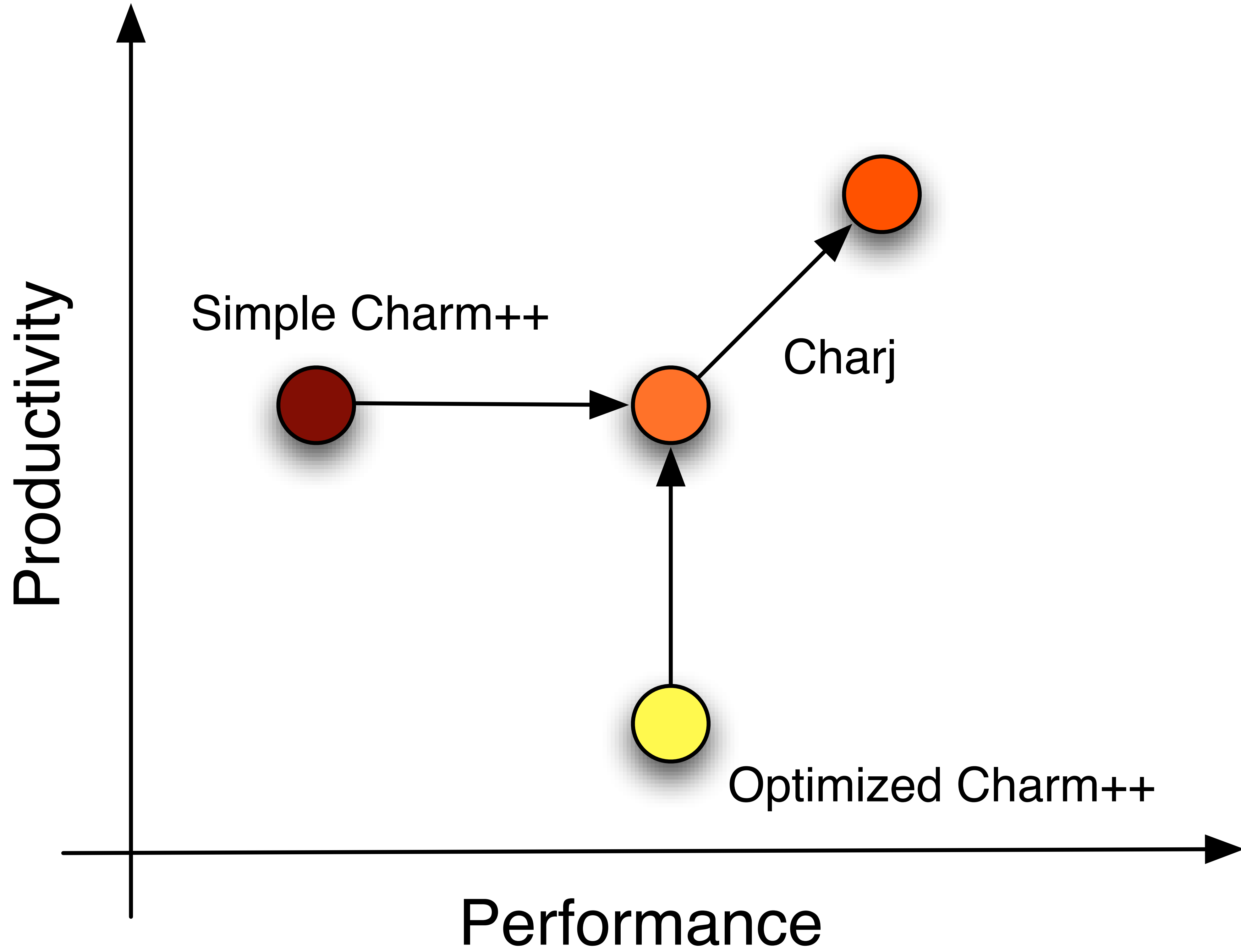
Thesis

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


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 sort of 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

```
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

Example: Jacobi

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

Jacobi: Message Driven Equivalent

```
entry void jacobi()
{
    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;
```

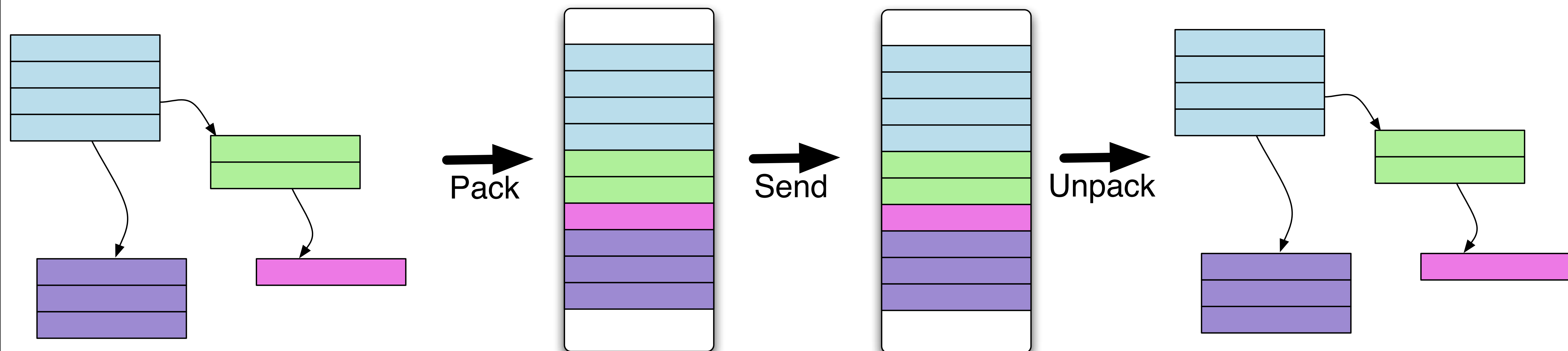
Accelerated Entry Methods

```
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 Applications

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