Optimizing for Productivity with Charj

Aaron Becker, PPL/UIUC

10th Annual Workshop on Charm++ and its Applications
8 May 2012

Productivity in High Performance Computing

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

Charj

More productive programs using the Charm programming model

- Make syntax more meaningful
- Apply static analysis to provide more safety and provide optimizations that are impossible at the library level
- Facilitate DSLs and “Little Languages”
- Add language-level support for rich runtime features

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

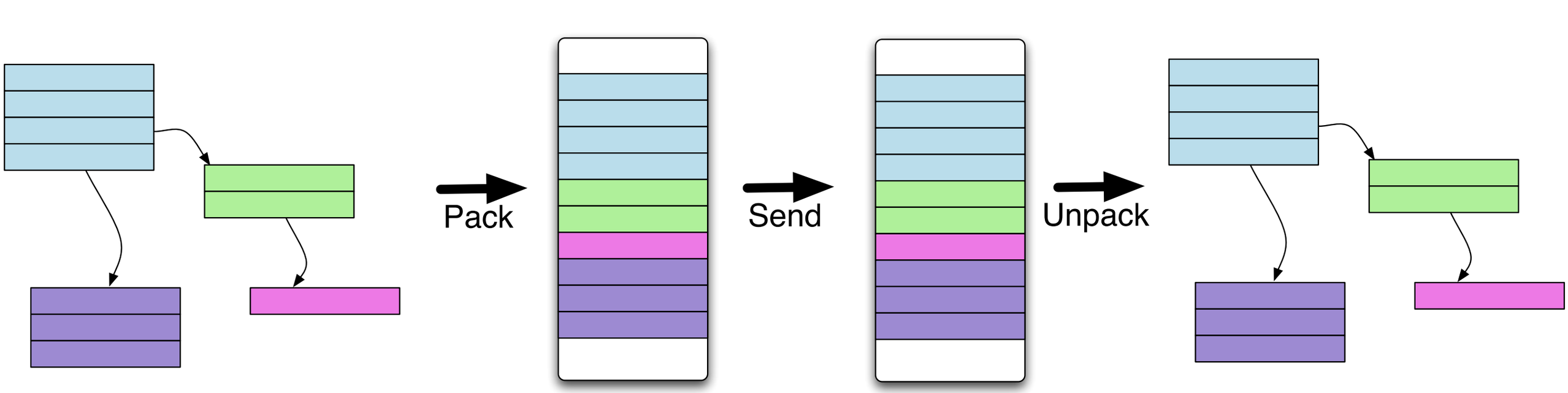
Productivity Benefits

- 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

Communication Optimization

Data Exchange

How do we communicate data structures in a parallel application?



Producing Pack/Unpack Functions

- Application data structures must be packed into and unpacked from buffers to be sent over the network
- MPI approach: user packs and sends the buffer
- Charm approach: user writes “PUP” method for each type describing how to pack and unpack it

Explicit Buffer Management

Send

```
memcpy(&var, buf, sizeof(var));
memcpy(&var2, buf+sizeof(var),
       sizeof(var2));
```

Receive

```
memcpy(buf, &var, sizeof(var));
memcpy(buf+sizeof(var), &var2
       sizeof(var2));
```

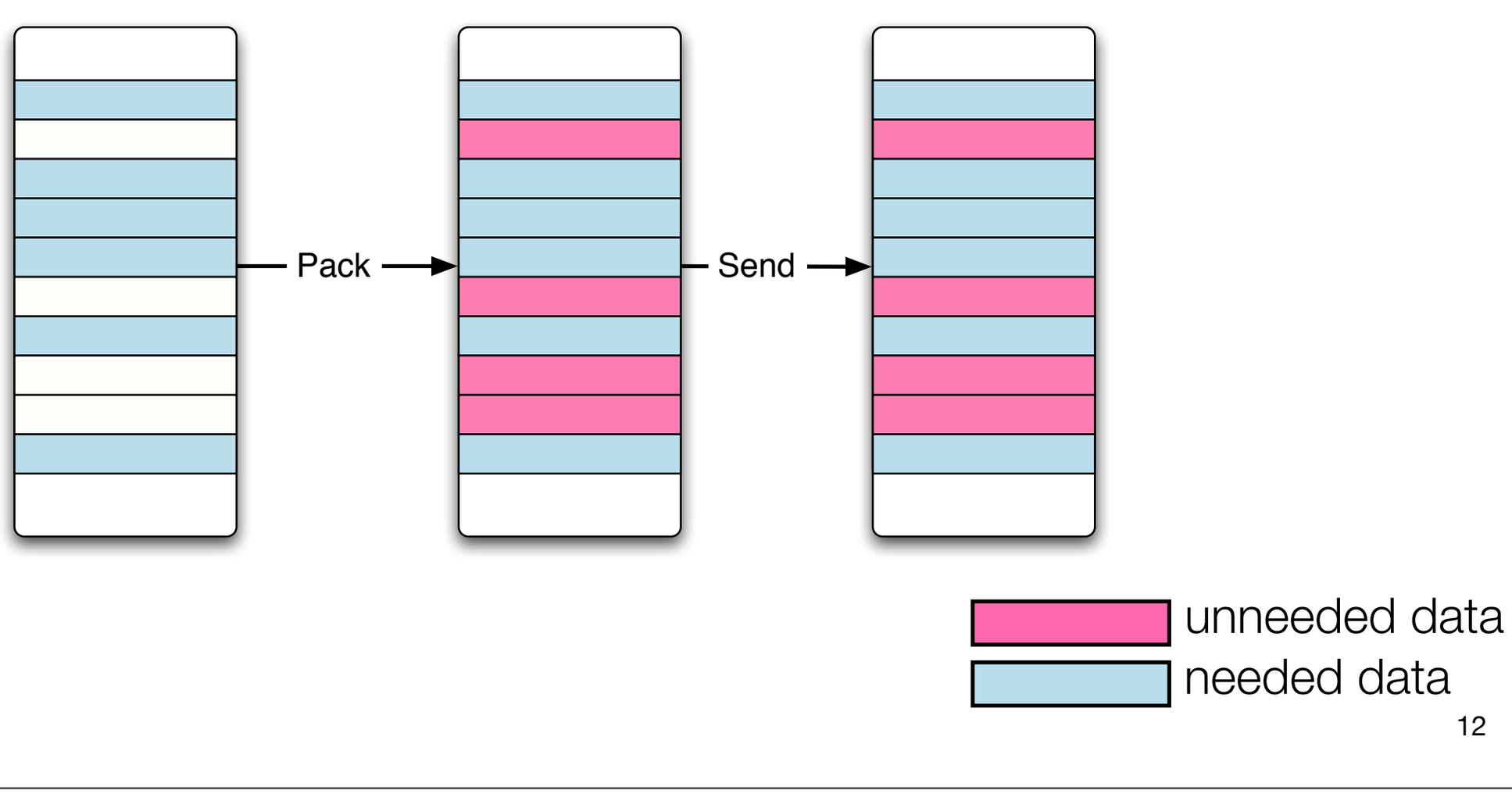
PUP (Pack/Unpack)

```
void MyType::pup(PUP::er& p)
{
    p | var1;
    p | var2;
}

entry void my_entry(MyType m);
```

Per-Method Pack/Unpack Functions

- In any method, some arguments may contain unneeded data
- For remotely invoked methods, this unneeded data hurts performance



Example: Molecular2D

```
class Particle {
    vec3 position;
    vec3 force;
    vec3 accel;
    vec3 vel;

    void pup(PUP::er& p) {
        p | position;
        p | force;
        p | accel;
        p | vel;
    }
};
```

```
class Compute {
    void interact(
        vector<Particle> remote_particles)
    {
        // only needs particle positions
    }
};
```

Example: Molecular2D

```
class Particle {
    vec3 position;
    vec3 force;
    vec3 accel;
    vec3 vel;

    void pup(PUP::er& p) {
        p | position;
        p | force;
        p | accel;
        p | vel;
    }
};
```

```
class Compute {
    void interact(
        vector<Particle> remote_particles)
    {
        // only needs particle positions
    }
};
```

Example: Molecular2D

```
class Particle {
    vec3 position;
    vec3 force;
    vec3 accel;
    vec3 vel;

    void pup(PUP::er& p) {
        p | position;
        p | force;
        p | accel;
        p | vel;
    }
};
```

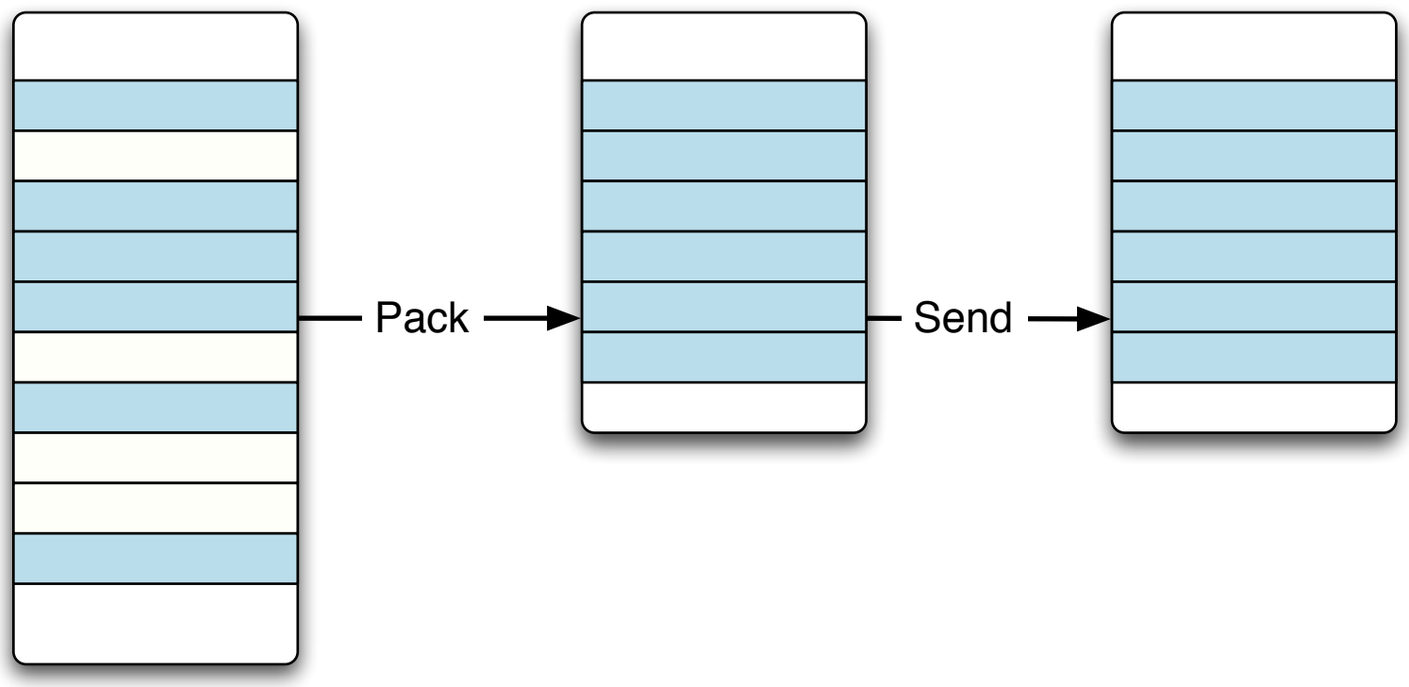
```
class Compute {
    void interact(
        vector<Particle> remote_particles)
    {
        // only needs particle positions
    }
};
```

75% of message is wasted!

Possible Solution

Use a custom type for the argument that only includes needed data

- Causes a proliferation of special-purpose, semantically meaningless types
- Scales poorly to large applications (lots of methods to handle)



Example: LeanMD

```
class Particle {
    vec3 position;
    vec3 force;
    vec3 accel;
    vec3 vel;

    void pup(PUP::er& p) {
        p | position;
        p | force;
        p | accel;
        p | vel;
    }
};
```

```
class Compute {
    void interact(ParticleDataMsg* m)
    {
        // only needs particle positions
    }
};
```

Example: LeanMD

```
class Particle {
    vec3 position;
    vec3 force;
    vec3 accel;
    vec3 vel;

    void pup(PUP::er& p) {
        p | position;
        p | force;
        p
        p
    }
};
```

```
class Compute {
    void interact(ParticleDataMsg* m)
    {
        // only needs particle positions
    }
};
```

Efficient communication, but what about time, effort, and lines of code?

Generating PUPs in Charj

- No need for user to provide PUP functions, since compiler knows the composition of each user-defined type
- Manage cyclic data structures by tracking memory addresses
- When the type is modified, the PUP function automatically changes with it, as opposed to Charm
- Programmer can override automatic PUP if needed

Live Variable Analysis

- Which fields are not needed on the receiving side?
    - Those which cannot be accessed in any path of execution
    - That is, anything that is not live at the function's preamble
- Live variable analysis can tell us which variables to pack and which to leave behind

Example: Charj

```
class Particle {
    vec3 position;
    vec3 force;
    vec3 accel;
    vec3 vel;
}
```

```
class Compute {
    entry void interact(
            Array<Particle> ps)
    {
        // only needs particle positions
    }
};
```

Smaller code, high efficiency

Productivity Benefits

- Programmer need not write PUP functions
- Type safe, avoids order dependence and pointer arithmetic bugs
- Per-method PUPs automatically strip out unneeded data
    - Creates smaller messages
    - Preserves meaningful types at the application level
    - Lets the programmer focus on higher-level performance issues instead of focusing on packing bytes into buffers

Questions?