The dHPF Compiler Project: Towards High Performance HPF

John Mellor-Crummey

Center for Research on Parallel Computation and
Department of Computer Science
Rice University

Theme: aggressive compiler optimization to make HPF approach hand-coded performance

Project Participants

Principal Designers

Vikram Adve, John Mellor-Crummey

Researchers

Robert Fowler, Ken Kennedy, Charles Koelbel

Graduate Students

Mark Anderson, Arun Chauhan, Chen Ding, Bo Lu,
Collin McCurdy, Dejan Mircevski, Mike Paleczny

Research Programmers

Don Baker, Kathi Fletcher, Monica Mevenkamp, Guohua Jin

Past Participants

Zoran Budlimic, Gil Hansen, Nenad Nedelkjovic,
Ajay Sethi, Lisa Thomas, Lei Zhou

Outline

- Motivation
- HPF Performance Issues
- dHPF Compiler Overview
- Recent Accomplishments
  - loop splitting
  - control simplification
  - in-place communication
- Ongoing Work and Plans

Motivation

HPF provides a convenient, portable parallel programming model

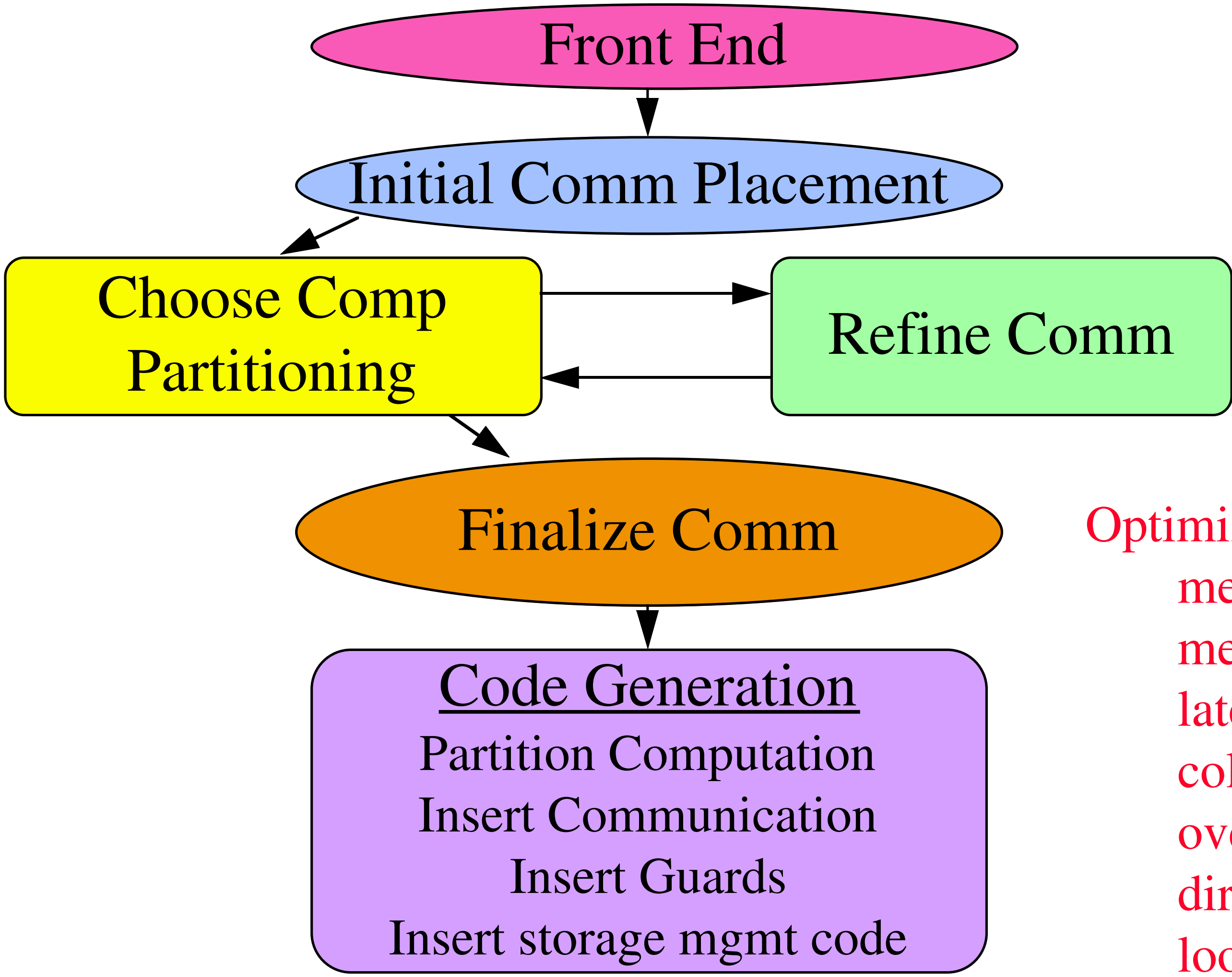
But ...

- Commercial compilers lag hand-coded performance
  - need better optimization for regular, irregular applications
- New architectures need new techniques
  - distributed shared memory (DSM - hardware & software)
  - clusters of SMPs
- New language features needed to expand applicability
  - computation on out-of-core arrays
  - advanced data structure support, e.g. pointers
  - more parallelism models, e.g. task-parallelism

HPF Performance Issues

- Unrealized parallelism
  - loops with dependences carried on few iterations
  - load imbalance
- Parallelization overhead
  - excess guards, data copying
  - dynamic memory allocation
- Excess communication
  - too many messages, too much data, redundant transfers
  - overly broad synchronization
- Memory locality
  - memory layout model
  - tiling, prefetching

dHPF Compiler Overview



Optimizations

- message vectorization
- message coalescing
- latency hiding
- collective communication
- overlap areas
- direct buffer access
- loop splitting
- copy minimization
- control simplification

Recent Accomplishments

- Unrealized parallelism
  - generalized loop splitting to improve parallelism
- Parallelization overhead
  - control simplification through constraint propagation
  - in-place communication to avoid data copying
  - static allocation of constant sized buffers
- Excess communication
  - support for min/minloc and max/maxloc combined reductions

Loop Splitting to Improve Parallelism

Input

```
do k = 1, N                                    ! sequential
    y(k,n) = (1.0 - x(k,1)) * y(1,n) + x(k,1) * y(n,n)
enddo
```

Goal

```
y(1,n) = (1.0 - x(1,1)) * y(1,n) + x(1,1) * y(n,n)
do k = 2, n-1                                  ! parallel
    y(k,n) = (1.0 - x(k,1)) * y(1,n) + x(k,1) * y(n,n)
enddo
y(n,n) = (1.0 - x(n,1)) * y(1,n) + x(n,1) * y(n,n)
```

Loop Splitting Strategy

Generalization of peeling transformations

- Identify carried data dependences
- When a carried dependence is for a loop-invariant location associated with the constant i=C, split loop into 3 sections
  - $i < C$
  - $i = C$
  - $i > C$
- Applies to nested loops with loop-invariant dependence endpoints with > 1 constant, e.g. x(3, j, 5)
- Mechanics: use integer set framework to perform complex splitting transformations, preserving the order of all operations

Control Simplification Example

Before

```
   do j = 1, 64
1      if (pid >= 1) then
2         k = 16 * pid + 1
3         if (16 * pid <= k - 2)
             COMPUTE f(1:64, j, k)
4         if (16 * pid == k - 1  && pid >= 1)
             RECV f(1:64, j, 16*pid)
5         if (16 * pid >= k - 1)
             COMPUTE f(1:64, j, k)
6      do k = 16 * pid + 2, min(16 * pid + 16, 63)
7         if (16 * pid == k - 17 .and. pid <= 2)
             SEND f(1:64, j, k)
8         if (16 * pid <= k - 2)
             COMPUTE f(1:64, j, k)
9         if (16 * pid == k - 1 && pid >= 1)
             RECV f(1:64, j, 16*pid)
10        if (16 * pid >= k - 1)
             COMPUTE f(1:64, j, k)
        if (pid <= 2)
11         k = 16 * pid + 17
12         if (16 * pid == k - 17 .and. pid <= 2)
             SEND f(1:64, j, k)
```

tautological

unsatisfiable

After

```
do j = 1, 64
1      if (pid >= 1) then
2         k = 16 * pid + 1
          RECV f(1:64, j, 16*pid)
       COMPUTE f(1:64, j, k)
6      do k = 16 * pid + 2, min(16 * pid + 16, 63)
          COMPUTE f(1:64, j, k)
        if (pid <= 2)
11        k = 16 * pid + 17
          SEND f(1:64, j, k)
```

Control Simplification Algorithm Sketch

Each conditional branch enforces some constraints

- Collect constraints at each control point in the program
  - incoming(x) = union over all paths reaching x
  - outgoing(x) = incoming(x) + local(x)
- Represent affine constraints as linear inequalities
- Apply Fourier-Motzkin elimination to determine
  - if constraints satisfiable at a control point
  - what constraints enforced at a control point are redundant
- Eliminate unsatisfiable conditionals
- Eliminate tautological conditional tests

Control Simplification Impact

6 experiments with 3 benchmarks (tomcatv, jacobi, erlebacher kernel)

- Eliminated 35-81% of conditionals (average 55%)
- Reduced overall code size 2-34% (average 22%)
- Improved execution time 1-16% (average 6%)
- Loop splitting + constraint simplification competitive with overlap areas (2% faster tomcatv; 2% slower jacobi; erle same)

In-Place Communication

When data to be sent in a message is contiguous,

no need to pack it in a buffer

(Surprisingly complex for a general integer set representation)

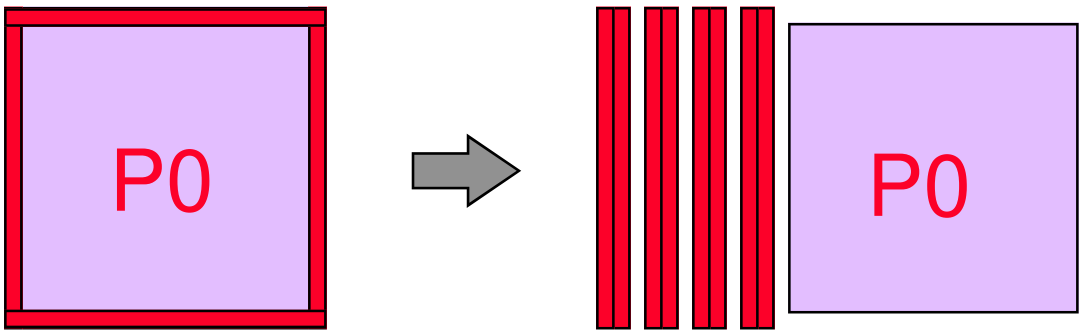
- Recognize n-dimensional contiguous data as that which
  - spans the array in leading dimensions 1 <= i <= k <= n
  - spans a contiguous range in dimension k
  - has a singleton index in dimensions k < j <= n
- Prove/Disprove at compile time using a sequence of tests
- When no compile-time determination is possible
  - generate code to evaluate unknown predicates
  - generate multi-version code to handle either outcome

Ongoing Work and Plans

- Extend code generation framework to directly target DSM
  - leverage message-passing compiler optimizations
- Continue investigation of compiler support for
  - out-of-core arrays
  - multi-block codes
  - irregular problems
- Explore scalar optimization effects for generated SPMD code

Ongoing Work: DSM Code Generation

- Explore memory model based on fragmented, distributed objects
  - separate shared data from non-shared data



    - » improves spatial locality for shared data
    - » provides uniform prefetch distance for data
  - avoid data copies by computing into buffers
  - use versioning to reduce cost of anti-dependences
- Explore algorithms for minimizing synchronization