Irregular Communication
Optimizations in PGAS

Jimmy Su and Katherine Yelick
Computer Science Division
UC Berkeley
Irregular Data Structures

- Irregular array accesses arise in many scientific applications.
  - Sparse matrix algorithms
  - Particle in cell methods
- Array access pattern is not known until runtime.
  - Example: particle affects its neighbor to the north where particle’s location is not known until runtime
  - When the fluid cell is remote, communication is required.
- Optimization space depends on architecture: cluster, multicore, GPU
Heart Simulation

- Problem: compute blood flow in the heart
- Performs well on large parallel machines
  - Scalable up to 512 processors
  - Close to achieving the 1 second per time step goal
- The code contains substantial amount of manual optimizations
Reasons for Manual Optimization

• Lack of trust in the compiler to make the right optimizations

• Programmers like to program in the way that they are most familiar with
  – Reluctant to learn a new parallel programming model without significant gain in productivity and/or performance.
Challenges in Heart Code

- Irregular data structures
- Communication optimizations
  - Packing and unpacking
  - Message aggregation and reduction
  - Atomic updates to fluid values
- Fiber partitioning
  - Balancing workload while minimizing communication
- Correctness debugging
Spread Force

2D Example

- Each particle spreads its force to its neighboring fluid cells
- A fluid cell may have multiple neighboring particles
- Updates to a fluid cell must be synchronized
SPMD Spread Force

for (int i=0; i<myParticleArray.length; i++){
    Particle p = myParticleArray[i];
    Point pos = [p.x, p.y];
    synchronize on fluidLock{
    }
}

Global locking overhead
Lock and unlock result in two network roundtrips on distributed backends

Many small messages
Each read or write may result in a network roundtrip if the memory location is remote
SPMD Spread Force with Manual Optimization

FluidCache cache = new FluidCache();
for (int i=0; i<myParticleArray.length; i++) {
    Particle p = myParticleArray[i];
    Point pos = [p.x, p.y];
    cache.add(pos+north, p.force);
}

send cache to the processor that owns the underlying fluid

barrier;

increment fluid using values from incoming caches

Reduces the number of messages by accumulating updates locally

Aggregates messages by sending all the updates for a remote processor together

Avoids global locking
Achieves synchronization by having the owner to do the increments serially
Automatic Optimization

Manual optimization significantly increases performance, but greatly increases the lines of code and makes debugging more difficult.

Goal: to achieve the manual optimizations automatically using the compiler.
Synchronized Region Analysis

Informally, we need to verify that the synchronization obtained by performing the updates serially at the remote processor is sufficient to replace the use of the global lock.

Performing updates serially at the remote processor gives us:

1. Atomic write to the shared variable
2. Makes the increments from different processors atomic with respect to each other

It does not:

1. Prevent data races between reads and writes to shared variables inside the synchronized region with
   a. Reads and writes outside the region protected by the same lock
   b. Reads and writes inside the same region
Synchronized Region Analysis

synchronized region

s1

write to shared variable

s2

no write to shared variable

s3

s4

Put in canonical form, it must be in one of the following forms:

Let $a$ be the shared variable

\[
\begin{align*}
    a &= expr \\
    a &= a \text{ } op \text{ } expr
\end{align*}
\]

where $a$ is not used in the evaluation of $expr$, and $op$ is an associative operator
Synchronized Region Analysis

Conditions 1 and 2 provide the following properties:

a. Updates to shared variables can be accumulated and delayed until the end.

b. Write to shared statements commute, so they can be executed in parallel.
Synchronized Region Analysis

$a$ and $b$ are shared variables

Write to shared statements can be executed in parallel, so $\text{expr1}$ and $\text{expr2}$ can be evaluated independently.

Partial order enforced by the remote processor that processes the updates serially.

partial order
Message Aggregation

Processor 1

<table>
<thead>
<tr>
<th>s1</th>
<th>s1</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3</td>
<td>s3</td>
</tr>
<tr>
<td>a=a op val1</td>
<td>a=a op val3</td>
</tr>
<tr>
<td>b=b op val2</td>
<td>b=b op val4</td>
</tr>
</tbody>
</table>

Processor 1 accumulates the updates locally

\[
a=a \text{ op } (\text{val1 op val3})
\]

\[
b=b \text{ op } (\text{val2 op val4})
\]

Update aggregation at each processor uses the associative property of \( op \).

Assume processor 1 works on \( n \) elements of the distributed array. Reduces \( 2n \) messages to 1.
Prevent Data Races

Performing updates serially at the remote processor does not:

1. Prevent data races between reads and writes to shared variables inside the synchronized region with
   a. Reads and writes outside the region protected by the same lock

Solution:

Have all processors start executing the optimized synchronized region together and exit together. This prevents all data races between reads and writes in the synchronized region and code outside of the region.

Titanium’s textually aligned barriers make analysis simple to determine whether the synchronized region will be executed by all processors.
Communication Methods

• Pack
  – Communicate only elements that are needed without duplicates.
  – List of array indexes
• Bound
  – Compute a bounding box that contains the needed data
  – Fill the gaps with identity elements
  – Communicate the box using one-sided bulk operation
• Multiple Bounding Boxes

Pack and bound are two extreme cases of multiple bounding boxes. Performance tuning is on whether to include a gap in the current bounding box or to start a new bounding box.
Performance Model

Use greedy algorithm to decide whether to include the gap in the current bounding box or start a new bounding box.

Cost of including the gap in the current bounding box:
\[ \text{gapSize} \times (\alpha + 1/b_w + 2\alpha) \]

Cost of starting a new bounding box:
\[ \text{descriptorSize} \times (\alpha + 1/b_w + \alpha) \]

\[ \alpha = \alpha_1 + \left(1/L_{\text{line}}\right)(\alpha_2 - \alpha_1) + \left(1/L_{\text{line}}\right)(\alpha_{\text{mem}} - \alpha_2) \]

\( b_w \) is the network bandwidth for a given size message.
Experimental Results

Experiments were done on a cluster of Opterons connected by Infiniband.
Message Reduction

Upper bound on the total number of remote messages

Without optimization:
Histogram: \( \text{sampleTotal} \cdot \frac{(p-1)}{p} \)
Particle gravitation: \( \text{particleNum}^2 \cdot \frac{(p-1)}{p} \)
Spread Force: \( \text{particleNum} \cdot \frac{(p-1)}{p} \)

With optimization:
At most two messages between each processor pair assuming each processor has some updates for every other processor

Since the input size is much larger than the number of processors, we see several orders of magnitude reduction in message count.
Conclusion

• Using automatic compiler optimization, we are able to achieve significant speedups on the fine grain code
  – Avoids global locking
  – Achieve aggregation by caching the updates locally
  – Overlap communication by sending non-blocking update requests to multiple remote processors
Textual Barrier Alignment

• Titanium has *textual barriers*: all threads must execute the same *textual* sequence of barriers
  – This example is illegal:
    ```java
    if (Ti.thisProc() % 2 == 0)
        Ti.barrier(); // even ID threads
    else
        Ti.barrier(); // odd ID threads
    ```

• Compiler proves where in the program a barrier can be inserted safely
  – We label those program points as in single context