Global-View Abstractions for User-Defined Reductions and Scans

PPoPP
March 29, 2006

Steve Deitz
Brad Chamberlain
David Callahan
Larry Snyder
A Bit of Context

- DARPA **HPCS**: High Productivity Computing Systems
  - HPLS: High Productivity Language Systems

- Increase productivity for HEC community by 2010

  \[ \text{Productivity} = \text{Programmability} + \]
  \[ \quad \text{Performance} + \]
  \[ \quad \text{Portability} + \]
  \[ \quad \text{Robustness} \]

- Revolutionary results (not evolutionary)

- Marketable to people other than program sponsors

- Phase II competition: Cray, IBM, and Sun
What is Chapel?

- **Chapel**: Cascade High-Productivity Language

- **Overall goal:**
  - Simplify the creation of parallel programs
  - Support evolution to production-grade codes
  - Emphasize generality

- **Motivating Language Technologies:**
  1) Global-view multithreaded parallel programming
  2) Locality-aware programming
  3) Object-oriented programming
  4) Generic programming and type inference
A Bit of Background (Reduce)

- Definition

The **reduce operator** takes a binary operation \( \oplus \) and a sequence of values \((a_1, a_2, \ldots, a_n)\) and returns the value

\[
a_1 \oplus a_2 \oplus \ldots \oplus a_n.
\]

- Example

\[\text{+ reduce (}1, 2, 3, 4, 5, 6, 7, 8, 9, 10\text{)}\]

\[\rightarrow 55\]
A Bit of Background (Scan)

◆ Definition

The **scan operator** takes a binary operation $\oplus$ and a sequence of values $(a_1, a_2, \ldots, a_n)$ and returns the sequence of values

$$(a_1, a_1 \oplus a_2, \ldots, a_1 \oplus a_2 \oplus \ldots \oplus a_n).$$

◆ Example

$+$ scan $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$

$\Rightarrow (1, 3, 6, 10, 15, 21, 28, 36, 45, 55)$
Parallel Performance

- Performance of reduce and scan
  - Highly efficient implementation
    - Log-tree for reduce
    - Parallel-prefix for scan
  - Associative operator → parallel performance
  - Commutative operator → more efficient

- Application of reduce and scan
  - Common in scientific computing
    - Used to test for convergence in iterative algorithms
    - Used in matrix multiplication and sorting kernels
    - ~9% of MPI calls in NPB are reductions (static count)
  - Supported on today’s systems
Outline

- Reduce and Scan Overview
- Local-View vs. Global-View Overview
- MPI and Chapel Abstraction for Reduce and Scan
- Application to MPI and Quantitative Results
Local vs. Global View

◆ **Local-View Programming:**
  - Programmer codes on a per-processor basis
    - Breaks distributed data structures into per-process chunks
    - Breaks work into per-process iterations/control flow

◆ **Global-View Programming:**
  - Programmer codes independent of processors (mostly)
    - Relies on compiler to manage decomposition details
    - Provides guide to decomposition at a high level
**Example**: “Apply a stencil to a vector”

\[
\text{global-view} \quad \begin{array}{c}
( & \begin{array}{cccc}
\text{Blue} & \text{Blue} & \text{Blue} & \text{Blue} \\
\text{Blue} & \text{Blue} & \text{Blue} & \text{Blue} \\
\text{Blue} & \text{Blue} & \text{Blue} & \text{Blue} \\
\text{Blue} & \text{Blue} & \text{Blue} & \text{Blue} \\
\end{array} \\
+ & \begin{array}{cccc}
\text{Blue} & \text{Blue} & \text{Blue} & \text{Blue} \\
\text{Blue} & \text{Blue} & \text{Blue} & \text{Blue} \\
\text{Blue} & \text{Blue} & \text{Blue} & \text{Blue} \\
\text{Blue} & \text{Blue} & \text{Blue} & \text{Blue} \\
\end{array} \\
= & \begin{array}{cccc}
\text{Blue} & \text{Blue} & \text{Blue} & \text{Blue} \\
\text{Blue} & \text{Blue} & \text{Blue} & \text{Blue} \\
\text{Blue} & \text{Blue} & \text{Blue} & \text{Blue} \\
\text{Blue} & \text{Blue} & \text{Blue} & \text{Blue} \\
\end{array}
\end{array}
\text{local-view} \quad \begin{array}{c}
( & \begin{array}{cccc}
\text{Yellow} & \text{Yellow} & \text{Yellow} & \text{Yellow} \\
\text{Yellow} & \text{Yellow} & \text{Yellow} & \text{Yellow} \\
\text{Yellow} & \text{Yellow} & \text{Yellow} & \text{Yellow} \\
\text{Yellow} & \text{Yellow} & \text{Yellow} & \text{Yellow} \\
\end{array} \\
+ & \begin{array}{cccc}
\text{Yellow} & \text{Yellow} & \text{Yellow} & \text{Yellow} \\
\text{Yellow} & \text{Yellow} & \text{Yellow} & \text{Yellow} \\
\text{Yellow} & \text{Yellow} & \text{Yellow} & \text{Yellow} \\
\text{Yellow} & \text{Yellow} & \text{Yellow} & \text{Yellow} \\
\end{array} \\
= & \begin{array}{cccc}
\text{Yellow} & \text{Yellow} & \text{Yellow} & \text{Yellow} \\
\text{Yellow} & \text{Yellow} & \text{Yellow} & \text{Yellow} \\
\text{Yellow} & \text{Yellow} & \text{Yellow} & \text{Yellow} \\
\text{Yellow} & \text{Yellow} & \text{Yellow} & \text{Yellow} \\
\end{array}
\end{array}
\]
Local vs. Global Example

◆ Example: “Apply a stencil to a vector”

**global-view**

```plaintext
var n: int = 1000;
var a, b: [1..n] float;

forall i in 2..n-1 do
    b(i) = (a(i-1) + a(i+1))/2;
```

**local-view**

```plaintext
var n: int = 1000;
var locN: int = n/numProcs;
var a, b: [0..locN+1] float;
var innerLo: int = 1;
var innerHi: int = locN;

if (iHaveRightNeighbor) {
    send(right, a(locN));
    recv(right, a(locN+1));
} else
    innerHi = locN-1;
if (iHaveLeftNeighbor) {
    send(left, a(1));
    recv(left, a(0));
} else
    innerLo = 2;
forall i in innerLo..innerHi do
    b(i) = (a(i-1) + a(i+1))/2;
```
3DStencil in NAS MG

```fortran
subroutine comm3(u,n1,n2,n3,kk)
    implicit none
    integer n1, n2, n3, kk
    integer axis
    double precision u( n1, n2, n3 )
    integer buff_len, buff_id

    indx = 0
    buff_id = 2 + dir
    buff_len = nm2
    do  i=1,nm2
        indx = indx + 1
        buff(i,buff_id) = 0.0D0
    enddo
    do  i1=1,n1
        indx = indx + 1
        buff(i1,2+dir) = u(i1,n2-1,
    do  i2=2,n2-1
        indx = indx + 1
        buff(i1,4) = buff(i1,3)
    enddo
    buff(idir,1) = buff(idir,2)
    buff(idir,2) = buff(idir,1)
    buff(idir,3) = buff(idir,4)
    buff(idir,4) = buff(idir,3)
    do  i3=2,n3-1
        indx = indx + 1
        buff(idir,12) = buff(idir,13)
        buff(idir,13) = buff(idir,14)
        buff(idir,14) = buff(idir,12)
        do  idir=1,4
            indx = indx + 1
            buff(idir,kk) = 0.0D0
        enddo
    enddo
    do  i1=1,n1
        indx = indx + 1
        buff(i1,14) = buff(idir, buff_id )
    enddo
    do  i2=2,n2-1
        indx = indx + 1
        buff(i1,12) = buff(idir, buff_id )
    enddo
    do  i3=2,n3-1
        indx = indx + 1
        buff(i1,10) = buff(idir, buff_id )
    enddo
    do  i1=1,n1
        indx = indx + 1
        buff(i1,11) = buff(idir, buff_id )
    enddo
    do  i2=2,n2-1
        indx = indx + 1
        buff(i1,15) = buff(idir, buff_id )
    enddo
    do  i3=2,n3-1
        indx = indx + 1
        buff(i1,16) = buff(idir, buff_id )
    enddo
    do  i1=1,n1
        indx = indx + 1
        buff(i1,17) = buff(idir, buff_id )
    enddo
    do  i2=2,n2-1
        indx = indx + 1
        buff(i1,18) = buff(idir, buff_id )
    enddo
    do  i3=2,n3-1
        indx = indx + 1
        buff(i1,19) = buff(idir, buff_id )
    enddo
enddo
enddo
enddo
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endi
Outline

◆ Reduce and Scan Overview
◆ Local-View vs. Global-View Overview
◆ MPI and Chapel Abstraction for Reduce and Scan
◆ Application to MPI and Quantitative Results
User Defined Reduction MinK

Compute the k minimum values in a sequence of values

E.g. mink(2) reduce (/8, 3, 5, 2, 7, 9, 10, 1, 4, 6/) → (1, 2)

E.g. mink(2) reduce (/Cray, Gonzaga, IBM, Sun, Duke/) → (Duke, Gonzaga)
Local vs. Global View

**Example:** “Reduce a sequence with Min2”

- **Global-view**
  
  \[(/8, 3, 5, 2, 7/)]
  
  \[(2, 3)]

- **Local-view**

  \[(/8, 3/)\]
  \[(/5, 2, 7/)\]

  \[(3, 8)\]
  \[(2, 5)\]

  \[(2, 3)\]

Local-view abstraction applies only to this part of the reduction.
MPI Reductions

◆ Has two varieties
  – Reduce returns reduction on root process
  – Allreduce returns result back to all processes

```
int MPI_Reduce(
    void* sendbuf,
    void* recvbuf,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm comm)
```

```
int MPI_Allreduce(
    void* sendbuf,
    void* recvbuf,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm comm)
```

◆ Supports nine built-in reduction operators
◆ Accepts arbitrary combine function
◆ MPI_Op assumes associativity, can be commutative
Local vs. Global View

◆ Example: “Reduce a sequence with Min2”

Global-view abstraction needs to hide the details of communication

Local-view abstraction applies only to this part of the reduction
Similar to classes in other OO languages
- Fields are accessed via "." syntax
- Methods are invoked "." syntax
- Support multiple inheritance, dynamic dispatch

Automatic default constructor

```chapel
class Point {
  var x : int, y : int;
}
class ColorfulPoint : Point {
  var c : color;
}
var cp = ColorfulPoint(x=2,y=3,c=red);
```

Default constructor provides named argument passing
Classes can be generic

class Triple {
    type elt_type;
    var x, y, z : elt_type;
}
def Triple.add(e : elt_type) {
    x += e; y += e; z += e;
}
var t = Triple(float);
t.add(1.0);
class reduction {
    type elt_type;
    param commutative = true;
    param associative = true;
    def accumulate(x : elt_type);
    def combine(s : reduction);
    def gen();
    def scan_gen(x : elt_type) do
        return gen();
    def red_gen() do
        return gen();
}

def reduce(r, s) {
    for e in s do
        r.accumulate(e);
    return r.red_gen();
}
class mink : reduction {
    var k : int;
    var v : [1..k] elt_type = max(elt_type);
    def accumulate(x : elt_type) do
        if x < v(1) {
            v(1) = x;
            for i in 2..k do
                if v(i-1) < v(i) then
                    (v(i-1),v(i)) = (v(i),(v(i-1));
        }
    def combine(s: mink(elt_type)) do
        for i in 1..k do
            accumulate(s.v(i));
    def gen() do
        return v;
}

... mink(2) reduce (/8, 3, 5, 2, 7/) ...
Outline

- Reduce and Scan Overview
- Local-View vs. Global-View Overview
- MPI and Chapel Abstraction for Reduce and Scan
- Application to MPI and Quantitative Results
Global-View in MPI

- **RSMPI (Reduce and Scan in MPI)**
  - Proof of concept of global view
  - Potential use for MPI programmers
- **Minimal changes to MPI to support global view**
  - Adds a construct to iterate over the local part of an ordered set
  - Uses a preprocessor to generate the accumulate loop based on the iterator construct.
  - Adds an operator as a collection of functions
    - Identity function
    - Accumulate function
    - Combine function
    - Generate function
  - Requires a single RSMPI_Reduce call
MG Initialization Results

Reduction: Find 10 minimum and 10 maximum values and locations

F+MPI: Uses 10 min and 10 max reductions
F+RSMPI: Uses one big reduction
Reduction: Determine if array is sorted

C+MPI: Compare boundary elements, determine if local portion is sorted, and ‘sum’ reduce local determinations

C+RSMPI: Use one big reduction
Summary

◆ High-level Chapel abstraction
  – Takes advantage of generic OOP capabilities
  – Contains both accumulate and combine function
  – Uses default constructor to produce identity
  – Supports generate function to return result that is different from the combining state
  – Supports a scan generate function that can return different result from a reduction

◆ RSMPI
  – Shows performance does not suffer at global view
  – Suggests path for incremental improvements

◆ Global-View
  – Orthogonal to changes in the distribution/layout
Questions?

For more information:

http://chapel.cs.washington.edu
email: chapel_info@cray.com