Lessons from Implementing the BiCGStab Method with SkeTo Library

Kiminori Matsuzaki\textsuperscript{1} and Kento Emoto\textsuperscript{2}

\textsuperscript{1} Kochi University of Technology
\textsuperscript{2} University of Tokyo
Agenda

Examine programmability and performance of skeletal parallelism with a real application

- BiCGStab method and target application
- SkeTo: parallel skeleton library
- Two approaches to development
- Experiment results
- Conclusion
BiCGStab Method

- Bi-Conjugate Gradient Stabilized method
  [van der Vorst, 1992]
  - Solve systems of linear equations: $Ax = b$
  - An iterative method
    - 2 matrix-vector multiplication (probmv function)
    - 4 inner-products
    - Other basic linear algebraic computations

```
function bicgstab(A, x, b, err, itrmax)
begin
    c := probmv(A, x)
p := s := r := b - c
    $\alpha := r \cdot r$
    if $(r \cdot r < err)$ then return $x$
    for it = 1 upto itrmax
        begin
            y := probmv(A, p)
            $\beta := s \cdot y$
            e := r - ($\alpha$/$\beta$)y
            v := probmv(A, e)
            $\gamma := (e \cdot v)/(v \cdot v)$
            x := x + ($\alpha$/$\beta$)p + $\gamma$e
            r := e - $\gamma$v
            if $(r \cdot r < err)$ then return $x$
            $\alpha := s \cdot r$
p := r + ($\alpha$/$($$\beta$$\gamma$))p - ($\alpha$/$\beta$)y
        end
    return $x$
end
```
Target Application

- Original program is given by a researcher
- BiCGStab is used in a 3D-space situation
  - The matrix-vector multiplication is actually a 3D-stencil computation

- A set of data is
  - a vector (in linear algebra)
  - a 3D-array (in stencil comp.)

```plaintext
for i = 0 upto I - 1
for j = 0 upto J - 1
for k = 0 upto K - 1
    c[i][j][k] = 
        a_p * b[i][j][k] 
        + a_e * b[i+1][j][k] + a_w * b[i-1][j][k] 
        + a_n * b[i][j+1][k] + a_s * b[i][j-1][k] 
        + a_t * b[i][j][k+1] + a_b * b[i][j][k-1] 
end
end
end
```
SkeTo: Parallel Skeleton Library

- A library with data-parallel skeletons
  - Data-parallel skeletons
    = collective operations on distributed data
  - Data structures: 1D-arrays (lists), 2D-arrays, trees

- Implemented in C++ and MPI
  - Originally for PC cluster computing
    ✓ Also available for recent multicore CPUs
  - SPMD model: but, parallelism is only in skeletons

- Optimization by fusion transformation
Parallel List Skeletons (1)

- (Almost) Element-wise computations
  - For $N$ elements, $O(N/P)$ time with $P$ processors.

\[
\begin{array}{cccc}
5 & 1 & 3 & 4 \\
\end{array} \quad \text{map (×2)} \quad \begin{array}{cccc}
10 & 2 & 6 & 8 \\
\end{array}
\]

\[
\begin{array}{cccc}
5 & 1 & 3 & 4 \\
\hline
2 & 5 & 1 & 3 \\
\end{array} \quad \text{zip (−)} \quad \begin{array}{cccc}
3 & -4 & 2 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
5 & 1 & 3 & 4 \\
\end{array} \quad \text{shift}_{\gg}(0) \quad \begin{array}{cccc}
0 & 5 & 1 & 3 \\
\end{array}
\]
Parallel List Skeletons (2)

- Reduction and scans (prefix-sums)
  - For $N$ elements, $O(N/P + \log P)$ time with $P$ procs.
  - Operators should be associative.

![Diagram showing reduction and scan operations on a list of numbers.](attachment:image.png)
Fusion Optimization

- One overhead of skeletal programs
  - Intermediate data between skeletons
  - E.g. \( \text{reduce}(+, \text{map}(^2, \text{map}(-\text{ave}, \text{as}))) \)
  - (2 intermediate arrays among 3 loops)

- Fuse skeletons and remove intermediate data
  - [IFL 2009]
  - By C++ template technique
  - Not only map and reduce, we can (partially) optimize scan or shift skeletons.
  - For simple programs like above, compiled code is as efficient as hand-written code.
Two Approaches

The first step to programming with SkeTo
Deciding how to map data in problems
to distributed structures in SkeTo

- **Target application involves:**
  - Linear algebraic computations (for 1D vectors)
  - 3D-stencil computations

- **Two approaches to implementation**
  - The whole vector $\rightarrow$ distributed list
  - 1D of 3D-space $\rightarrow$ distributed list
1st approach:

mapping the whole vector to list

- 3D-stencil computation
  - First, we flatten the nested loop.
  - We need to access originally neighbor (in 3D) but far-away elements (e.g. \( b[i][j+1][k] \))
  - Implement an extended shift skeleton: \( \text{gshift} \)

  ✓ As a user-defined skeleton
  
  ```
  function gshift(e, \( d \), \( [a_0, a_1, \ldots, a_d] \))
  begin
  return \( [e, \ldots, e, a_0, a_1, \ldots, a_{n-1-d}] \)
  end
  ```

- Linear algebraic computation
  - Easy to implement
Code for Stencil Computation

```cpp
*c = sl::zipwith(timesd, f,
    sl::zipwith(plusd, sl::map(std::bind1st(timesd, ap), b),
    sl::zipwith(plusd, sl::map(std::bind1st(timesd, aw), sl::shiftr(0.0, b)),
    sl::zipwith(plusd, sl::map(std::bind1st(timesd, ae), sl::shiftl(0.0, b)),
    sl::zipwith(plusd, sl::map(std::bind1st(timesd, as), gshift(+dy, 0.0, b)),
    sl::zipwith(plusd, sl::map(std::bind1st(timesd, an), gshift(-dy, 0.0, b)),
    sl::zipwith(plusd, sl::map(std::bind1st(timesd, ab), gshift(+dz, 0.0, b)),
    sl::zipwith(plusd, sl::map(std::bind1st(timesd, at), gshift(-dz, 0.0, b))))))));
```

- Skeletal code corresponds to algebraic definition
- Skeletons except for gshift can be optimized by fusion
2nd Approach:

Mapping 1D of 3D-space to list

- Slice 3D-space and define a list of planes
  ```
  struct array2D {
    double data[jmax+2][imax+2];
  }
  ```

- 3D stencil computation
  - We can implement with map, zip, and shift

- Linear algebraic computation
  - We need to define element-wise +, *, etc.
    - Function objects with a 2-nested loop
  - → additional 70 lines of code
Manual Fusion Transformation

- But, the 2nd one ran slow after fusion
  - E.g. For computing \( e := r - (\alpha/\beta)y \)

```c
for (i = 0; i < local_size; i++) {
    array2D tmp;
    for (int j = 1; j <= jmax; j++) {
        for (int i = 1; i <= imax; i++) {
            tmp[j][i] = alp * y[i][j][i];
        }
    }
    for (int j = 1; j <= jmax; j++) {
        for (int i = 1; i <= imax; i++) {
            e[i][j][i] = r[i][j][i] - tmp[j][i];
        }
    }
}
```

- Solutions: nested optimization, 3D-array skeletons, manual fusion (+50 lines)
Further Optimization

- The program still had large overhead.
- **Reason:** Elements are copied in skeletons
  - Design for simplicity (and fast for simple elements)
  - **Type of function objects:** `B operator() (const A&)`;
- **Our solution:**
  - Use smart pointers to avoid data copies; and a special implementation of skeletons with serialization
  - Serialization will be imported in SkeTo ver1.10
Experiment 1: On PC Cluster with Multicore-CPUs

- Relative speedup: 1D (1st) > 3D (2nd)
- Performance: Depend on size, #cores

![Graph showing relative speedup and performance for 1D and 3D computations on a PC cluster with multicore CPUs.](image)
Experiment 2:

On Dual-quadcore Server

- Performance: 3D (2nd) > 1D (1st)
- Overhead w.r.t. sequential programs (1D)
Experiment 3: Comparing Two Compilers

- Performance depends on approaches and compilers
  - Faster 1D code by GCC
  - Faster 3D code by Intel Compiler
Conclusion

- We have obtained 7 lessons (in the paper) in implementing the BiCGStab with SkeTo
  - Two implementation with list skeletons
    - By mapping the whole vector to list
    - By mapping 1D of 3D-space to list
  - Performed optimization with fusion transformation automatically/manually
  - Several experiment results are shown
    - Performance depends on problem-size, architecture, and compiler
Future Work

- Implement other real applications and examine what we need more
  - Application: Machine Learning, etc.
  -Skeletons: permute, groupByKey

- SkeTo ver. 1.10 coming soon
  http://www.ipl.t.u-tokyo.ac.jp/sketo/
  - Re-implementation of matrix skeletons
  - Support for serialization of user-defined data
  - (Experimental) C++0x support