Parallel Skeletons for Variable-Length Lists in SkeTo Skeleton Library

Aug. 27, 2009
Haruto Tanno
Hideya Iwasaki

The University of Electro-Communications (Japan)
Outline

• Introduction
• Problems of Exiting Fixed-Length Lists
• Proposed Variable-Length Lists
  – Skeletons and Operations
  – Data Structure
• Experiments
• Related Work
• Conclusion
Introduction

• Writing efficient parallel programs is difficult
  – synchronization, inter-process communications, data distributions among processes
• Solution: Skeletal parallel programming
  – implements generic patterns within parallel programs
    • C++ parallel skeleton library SkeTo [06] (intended for distributed environments such as PC cluster)
  – enables users to write parallel programs as if they were sequential
SkeTo

• SkeTo (Skeletons in Tokyo)
  – Constructive parallel skeleton library (C++ and MPI)
  – Joint project of The University of Tokyo, National Institute of Informatics, and The University of Electro-communications
  – Started from 2003, Version 1.0 coming soon

• Distinguishing features of SkeTo
  – It is based on the theory of Constructive Algorithmics
  – It provides skeletons for lists, matrices, and trees
  – It introduces no special extension to the base C++
Examples of List Parallel Skeletons

• map skeleton on list
  – applies a function to all elements

map \text{sqr} \ [1,2,3,4,5,6,7,8]
= [1,4,9,16,25,36,49,64]

Node #0 \hspace{1cm} Node #1 \hspace{1cm} Node #2 \hspace{1cm} Node #3

\begin{align*}
1 & \rightarrow \text{sqr} & 3 & \rightarrow \text{sqr} & 5 & \rightarrow \text{sqr} & 7 & \rightarrow \text{sqr} \\
2 & & 4 & & 6 & & 8 & \\
1 & \rightarrow & 4 & & 9 & \rightarrow & 16 & \\
& & & & 25 & \rightarrow & 36 & \\
& & & & 49 & \rightarrow & 64 & \\
\end{align*}

• reduce skeleton on list
  – collapses a list with an associative binary operator

reduce \ (+) \ [1,2,3,4,5,6,7,8]
= 1+2+3+4+5+6+7+8 = 36

Node #0 \hspace{1cm} Node #1 \hspace{1cm} Node #2 \hspace{1cm} Node #3

\begin{align*}
1 & \rightarrow + & 3 & \rightarrow + & 5 & \rightarrow + & 7 & \rightarrow + \\
2 & & 4 & & 6 & & 8 & \\
3 & \rightarrow + & 7 & \rightarrow + & 11 & \rightarrow + & 15 & \\
10 & \rightarrow + & 26 & & & & & \\
36 & & & & & & & \\
\end{align*}
Problem in Existing List

- Computing twin primes
  (Pairs of prime numbers that differ by two)

```plaintext
ums = [2,3,...,13,14,15], ps = [];
do { // the Sieve of Eratosthenes
    p = take the front element from nums
    remove every element that is divisible by p from nums
    add p to ps
} while( p <= sqrt_size );
concatenate ps and nums
// ps is [2,3,5,7,11,13]
twin_ps = make pairs of adjacent prime numbers
remove every pair of prime numbers whose difference isn’t two
// twin_ps = [(3,5), (5,7), (11,13)]
```

We can’t shrink or stretch lists because their size is fixed
Problems that Need Variable-Length Lists

Type 1: Problems that leave such elements in a given list that satisfy various conditions
- e.g., twin-primes problem, problem of the convex hull

Type 2: Searching problems in which the number of candidates for solutions may dynamically change
- e.g., knight’s tour

Type 3: Iterative calculations in which computational loads for all elements in a list lack uniformity
- e.g., calculations of Mandelbrot and Julia sets

If we can remove elements that have already finished their calculations, we can solve these problems efficiently
Our Proposal

Proposal
• We propose parallel skeletons for lists of variable lengths that enable us to solve a wide range of problems

Approach
• We implement a new library of variable-length lists in SkeTo (They are compatible with existing lists)
  – Provide new skeletons and operations that dynamically and destructively change a list’s length
  – Change data structure that expresses lists
  – Add automatic data relocation mechanism for adequate load balancing
Proposed Skeletons and Operations

- **concatmap** applies a function to every element and concatenates the resulting lists
  e.g., concatmap dup [1,2] => [1,1,2,2]
- **filter** leaves elements that satisfy a Boolean function
  e.g., filter odd [1,2,3,4,5,6,7,8] => [1,3,5,7]
- **append** concatenates two lists
  e.g., append [1,2] [3,4,5,6] => [1,2,3,4,5,6]
- **popfront** and **popback** remove an element from the front and the back in a list
- **pushfront** and **pushback** add an element to the front or the back in a list
C++ Program for Twin Prime Problem

...  
do{  // the Sieve of Eratosthenes  
p = nums->pop_front();  
list_skeletons::filter_ow(IsNotMultipleOf(p), nums);  
ps->push_back(p);  
} while(p <= sqrt_size);  
ps->append(nums);  
// ps is list of prime numbers  
dist_list<int>* dup_ps = ps->clone<int>();  
dup_ps->pop_front();  
twin_ps = list_skeletons::zip(ps, dup_ps);  
list_skeletons::filter_ow(twin_ps, IsTwin());  
// twin_ps is solution list
C++ Program for Knight’s Tour

dist_list<Board>* bs; // list of solution boards
...

// add initial board state to bs
bs->push_back(initBoard);

// increase bs dynamically up to MAXSIZE
while( bs->get_size() < MAXSIZE ){
    // generate next moves from each board states
    concatmap_ow(nextBoard, bs);
}

// search all solutions with depth first order
concatmap_ow(solveBoard, bs);

// bs is a list of solution boards
C++ program for Mandelbrot set

dist_list<point>* ps;       // list of points
dist_list<point>* rs;       // list of calculation results

... for ( int i=0; i<maxForCount; i++ ){  
    // progress calculations in small amounts
    map_ow(calc, ps);
    // remove elements that have already finished calculation
    dist_list<point>* es = filtersplit_ow(isEnd, ps);
    // add them to rs
    rs->append(es);
    delete es;
}
rs->append(ps);
// rs is a list of calculation results
Data Structure of Existing Fixed-Length List

- Elements in lists are equally distributed to each node using block placement
- Data placement does not change during computation

A = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
Data Structure of Variable-Length List (1)

- Each node has to know the latest information on the numbers of elements in other nodes

A = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]

B = [1, 2, 4, 5, 7, 8, 10, 11, 13]

If the numbers of elements on each node become too unbalanced, the data in list are automatically relocated.
Data Structure of Variable-Length List (2)

- When we concatenate two lists, we adopt block-cyclic placement without relocating the entire amount of data.

\[ A = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] \]
\[ B = [14, 15, 16, 17, 18, 19, 20] \]
\[ C = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20] \]

**A** and **B** are concatenated to form **C**.

Node #0

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Node #1

<table>
<thead>
<tr>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Block count = 1

**Block-cyclic placement**

**Concatenate A and B**

append

Cyclic placement of the concatenated list.
Micro-benchmark (1)

• Experimental environment
  – Pentium4 3.0GHz, Mem 1GB, Linux 2.6.8, 1000BaseT Ethernet

• We measured the execution times for applying map, reduce, and scan, and the data relocation in a list
  – Input list: 80 million elements
  – block count: from 1 to 4,000
  – two functions: short/long execution time
## Micro-benchmark (2)

### Execution time (s)

<table>
<thead>
<tr>
<th>block count</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>2000</th>
<th>3000</th>
<th>4000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map (short)</td>
<td>0.0128</td>
<td>0.0129</td>
<td>0.0128</td>
<td>0.0129</td>
<td>0.0129</td>
<td>0.013</td>
<td>0.0132</td>
</tr>
<tr>
<td>Reduce (short)</td>
<td>0.0183</td>
<td>0.0182</td>
<td>0.0183</td>
<td>0.0194</td>
<td>0.0194</td>
<td>0.0200</td>
<td></td>
</tr>
<tr>
<td>Scan (short)</td>
<td>0.0407</td>
<td>0.0408</td>
<td>0.0411</td>
<td>0.0484</td>
<td>0.053</td>
<td>0.0580</td>
<td></td>
</tr>
<tr>
<td>Map (long)</td>
<td>16.8</td>
<td>16.8</td>
<td>16.8</td>
<td>16.8</td>
<td>16.8</td>
<td>16.8</td>
<td>16.8</td>
</tr>
<tr>
<td>Reduce (long)</td>
<td>16.9</td>
<td>16.9</td>
<td>16.9</td>
<td>16.9</td>
<td>16.9</td>
<td>16.9</td>
<td>17.0</td>
</tr>
<tr>
<td>Scan (long)</td>
<td>33.8</td>
<td>33.8</td>
<td>33.8</td>
<td>33.9</td>
<td>34.3</td>
<td>34.7</td>
<td>34.1</td>
</tr>
<tr>
<td>Data Relocation</td>
<td>-</td>
<td>3.74</td>
<td>4.64</td>
<td>4.67</td>
<td>4.66</td>
<td>4.62</td>
<td>4.72</td>
</tr>
</tbody>
</table>

The overheads of data relocation are large

⇒ It is effective to delay the relocation of data
Macro-benchmark (1)

• Type1
  – Twin primes (list of 10 million integers)
  – Gift-wrapping method (1 million points)

• Type2
  – Knight’s tour (5 × 6 board)

• Type3
  – Mandelbrot set (1,000 × 1,000 coordinates)
    • Using variable-length lists with 100 iterative calculations
      × 100 times
    • Using fixed-length lists with 10,000 iterations
These results indicate excellent performance in all problems.
Macro-benchmark (3)

Programs with variable-length lists show good speedups
Parallel Skeleton Libraries

• P3L [95], Muesli [02], Quaff [06]
  – support data parallel skeletons
  – offer lists (distributed one-dimensional arrays)

• eSkel [05]
  – supports task parallel skeletons but does not support data parallel skeletons for list like map and reduce

• Muskel [07], Calcium [07]
  – a Java skeleton library on a grid environment

These libraries do not support variable-length lists
Another Group of Libraries

• **STAPL [07]**
  – Offers variable-length arrays and lists (pVector, pList)
  – Provides the same operations as C++ STL
  – Does not have operations such as `concatmap`

• **Data Parallel Haskell [08]**
  – Offers distributed nested lists
  – Provides `filter`, `concatmap`, and `append`
  – Only targets shared-memory environments
Conclusion

• We proposed parallel skeletons for variable-length list and their implementation
  – We proposed skeletons and operations for variable-length list, e.g., concatmap, filter, and append
  – We adopted a block-cyclic representation of lists with size tables
  – We confirmed the efficiency of our implementation through tests in various experiments
Data relocation (1)

<table>
<thead>
<tr>
<th>source</th>
<th>dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0</td>
<td>#0</td>
</tr>
<tr>
<td>#0</td>
<td>#1</td>
</tr>
<tr>
<td>#1</td>
<td>#1</td>
</tr>
<tr>
<td>#0</td>
<td>#1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**local data copy**

**inter node communication**
Data relocation (2)

- Making schedule
  - Begin non blocking inter node communication
    - Local data copy
    - Wait non blocking inter node communication
      - Transferring data
Cocatmap

A = [0, 1, 2, 3, 4, 5]

B = [0, 0, 1, -1, 2, -2, 3, -3, 4, -4, 5, -5]

Apply function plusminus

cocatmap

Node #0

0 1 2
3 3

Node #1

3 4 5
3 3

B = [0, 0, 1, -1, 2, -2, 3, -3, 4, -4, 5, -5]

Apply function plusminus
Pushback

A=[0,1,2,3,4,5]  

A=[0,1,2,3,4,5,6]
Condition of Data Relocation

\[ \text{imbalance}(A) = n \times \max(P_1, \ldots, P_n) / (P_1 + \cdots + P_n) \]

where \( P_i = \sum_{j=1}^{m} p_{ij} \), \( n \) = the number of nodes, \( m \) = the block count of \( A \),
\( p_{ij} \) = the number of elements of the \( j \)-th block at the \( i \)-th node.

\[ \text{imbalance}(A) > 1.5 \]

\( \Rightarrow \) data relocation
Convex hull

- Gift wrapping method
## Scan in Variable-Length List

<table>
<thead>
<tr>
<th>Node #0</th>
<th>Node #1</th>
<th>Node #2</th>
<th>Node #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 1 1</td>
<td>1 1 2</td>
<td>2 2 2</td>
</tr>
<tr>
<td>3 3 3</td>
<td>3 4 4</td>
<td>4 4</td>
<td>5 5</td>
</tr>
</tbody>
</table>

**local reduce**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 1 2</td>
<td>1 2 4</td>
<td>2 4 6</td>
</tr>
<tr>
<td>3 6 9</td>
<td>3 7 11</td>
<td>4 8</td>
<td>5 10</td>
</tr>
</tbody>
</table>

**sharing local reduce among nodes**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 1 2</td>
<td>1 2 4</td>
<td>2 4 6</td>
</tr>
<tr>
<td>3 3 3</td>
<td>3 7 11</td>
<td>4 8</td>
<td>5 10</td>
</tr>
</tbody>
</table>

**local scan**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 1 2</td>
<td>1 2 4</td>
<td>2 4 6</td>
</tr>
<tr>
<td>3 3 3</td>
<td>3 7 11</td>
<td>4 8</td>
<td>5 10</td>
</tr>
</tbody>
</table>

**calculating final scan**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 add 0</td>
<td>3 4 6 add 2</td>
<td>8 10 12 add 6</td>
<td></td>
</tr>
<tr>
<td>15 15 15 add 12</td>
<td>24 28 32 add 21</td>
<td>36 40 add 32</td>
<td>45 50 add 40</td>
</tr>
</tbody>
</table>
Reduce in Variable-Length Lists

<table>
<thead>
<tr>
<th>Node #0</th>
<th>Node #1</th>
<th>Node #2</th>
<th>Node #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>011</td>
<td>112</td>
<td>222</td>
</tr>
<tr>
<td>333</td>
<td>344</td>
<td>44</td>
<td>55</td>
</tr>
</tbody>
</table>

local reduce

<table>
<thead>
<tr>
<th>000</th>
<th>011</th>
<th>112</th>
<th>222</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

collapse result value to Node #1

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>11</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

broadcast result

| 50     | 50     | 50     | 50     | 50     | 50     |

12 + 38 = 50
## Zip in Variable-Length Lists

<table>
<thead>
<tr>
<th>Node #0</th>
<th>Node #1</th>
<th>Node #2</th>
<th>Node #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5</td>
<td>6 7</td>
<td>13 14</td>
<td>15 16</td>
</tr>
<tr>
<td>8 9 10</td>
<td>11 12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Node A**

<table>
<thead>
<tr>
<th>Node #0</th>
<th>Node #1</th>
<th>Node #2</th>
<th>Node #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4</td>
<td>5 6 7 8</td>
<td>9 10 11 12</td>
<td>13 14 15 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Node B**

<table>
<thead>
<tr>
<th>Node #0</th>
<th>Node #1</th>
<th>Node #2</th>
<th>Node #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4</td>
<td>5 6 7 8</td>
<td>9 10 11 12</td>
<td>13 14 15 16</td>
</tr>
</tbody>
</table>