Parallel Skeletons for Variable-Length Lists in SkeTo Skeleton Library

Haruto Tanno and Hideya Iwasaki

The University of Electro-Communications
1-5-1 Chofugaoka, Chofu, Tokyo 182–8585 Japan
tanno@ipl.cs.uec.ac.jp, iwasaki@cs.uec.ac.jp

Abstract. Skeletal parallel programming is a promising solution to simplify parallel programming. The approach involves providing generic and recurring data structures like lists and parallel computation patterns as skeletons that conceal parallel behaviors. However, when we focus on lists, which are usually implemented as one-dimensional arrays, their length is restricted and fixed in existing data parallel skeleton libraries. Due to this restriction, many problems cannot be coded using parallel skeletons. To resolve this problem, this paper proposes parallel skeletons for lists of variable lengths and their implementation within a parallel skeleton library called SkeTo. The proposed skeletons enable us to solve a wide range of problems including those of twin primes, Knight's tour, and Mandelbrot set calculations with SkeTo. We tested and confirmed the efficiency of our implementation of variable-length lists through various experiments.

1 Introduction

Writing efficient parallel programs is difficult, because we have to appropriately describe synchronization, inter-process communications, and data distributions among processes. Many studies have been devoted to this problem to make parallel programming easier. Programming with parallel computation patterns (or skeletons), or skeletal parallel programming [8,10,15] in short, has been proposed as one promising solution. Skeletons abstract generic and recurring patterns within parallel programs and conceal parallel behaviors within their definitions. They are typically provided as a library. Skeletons can be classified into two groups: *task parallel* skeletons and *data parallel* skeletons. Task parallel skeletons capture the parallelism by the execution of several different tasks, while data parallel skeletons capture the simultaneous computations on the partitioned data among processors. Parallel skeleton libraries enable users to develop a parallel program by composing suitable skeletons as if it were a sequential program.

However, in data parallel skeletons offered by existing parallel skeleton libraries, the length of a list, which is usually implemented as an one-dimensional array, is fixed due to an implementation reason. Thus, operations that dynamically and destructively change a list's length without allocating a new list within a memory area, e.g, adding elements to a list or removing elements from it, are not permitted. As a result, some problems cannot be solved efficiently or cannot be easily or concisely described by using existing data parallel skeletons.

```
numbers = [2,3,4,5,6,...,50]  ; // list of numbers
primes = [];                    // list of prime numbers
twin_primes = [];               // list of twin primes
// make a list of primes using the Sieve of Eratosthenes
do {
    prime = take front element from numbers
    remove elements that are divisible by prime from numbers
    add prime to primes
} while ( prime <= sqrt(MAX) );
concatenating primes and numbers
// primes is [2,3,5,7,11,13,...,47]
// make a list of twin primes from a list of primes
twin_primes = making pair of adjacent prime numbers
// twin_primes is [(2,3),(3,5),(5,7),(7,11),(11,13),...,(43,47)]
remove each pair of prime numbers whose difference is not two
// twin_primes is [(3,5),(5,7),(11,13),(17,19),(29,31),(41,43)]
```

Fig. 1. Pseudo-code for computing twin primes

For example, consider a problem to compute twin primes. Twin primes are pairs of prime numbers that differ by two, e.g., $(3, 5)$ and $(5, 7)$. A pseudo-code for this problem that computes twin primes of less than or equal to fifty is shown in Fig. 1. First, we create a list of prime numbers from a list of integers by using the Sieve of Eratosthenes. Second, we create a list of pairs of adjacent prime numbers. Finally, we remove every pair whose difference is not two to obtain a list of twin primes. A list of fixed length does not offer operations such as removing elements from a list, adding elements to it, or concatenating two lists, since these operations obviously change the length of the list. If a list of variable length (a list whose length may dynamically change) were supported, we could solve this problem based on the concise code in Fig. 1.

We could emulate a variable-length list by using a fixed-length list whose elements were pairs of a Boolean value and an integer. The Boolean value in an element indicates whether the element is actually contained in the list or not. By using such a list, we would be able to obtain a list of prime numbers with existing skeletons for fixed-length lists. However, we cannot make a list of pairs of adjacent prime numbers, because adjacent Boolean-integer pairs are not always adjacent prime numbers. Furthermore, we are forced to write complicated code due to the manipulation of Boolean-integer pairs.

The restriction that the length of a list is fixed mainly comes from an efficiency reason. In existing skeleton libraries for distributed environments, elements in a list are statically distributed at its initialization time. This enables these libraries to be efficiently implemented without extra communications between nodes. The most straightforward way for removing this restriction is to provide new list skeletons, each of which produces within a memory area an output list whose length is different from that of an input list. However, these skeletons are inefficient in many cases, because they need extra memory allocations. Thus, we want new list skeletons with the following features.

– To avoid inefficient memory usage, each skeleton destructively reuse the memory occupied by its input list for its output list that may have different length.
– To avoid inefficient data communications that dissolve uneven data distributions of elements of a list, dynamic and flexible data re-distribution mechanism is necessary in the implementations of the new skeletons.

In this paper, we propose data parallel skeletons for variable-length lists and implement them within a parallel skeleton library called SkeTo [14,16]. The skeletons for variable-length lists enable us to solve a wide range of problems with SkeTo.

The research discussed in this paper makes three main contributions.

1. We analyzed typical patterns of problems that needed variable-length lists and determined which skeletons and operations would be supported in the SkeTo library. Although the new skeletons and operations that were supported were quite common, especially within the community of functional programming languages, they should suffice to describe programs for typical problems.
2. We demonstrated that a block-cyclic representation of a list with a size table effectively enabled a variable-length list to be implemented. This representation was flexible enough to cope with changes in the list's length, and enabled the library to delay the relocation of elements in a list whose elements were unevenly distributed between nodes.
3. Through tests in various experiments, we confirmed the efficiency of our implementation of variable-length lists.

2 Existing List Skeletons in SkeTo

SkeTo (Skeletons in Tokyo) [14,16] is a parallel skeleton library intended for distributed environments such as PC clusters, in which all machines (called *nodes* after this) in the cluster are composed of a single-core CPU. It has three distinguishing features.

1. It provides a set of data parallel skeletons that are based on the theory of Constructive Algorithmics [4] for recursively-defined data structures and functions.
2. It enables users to write parallel programs as if they were sequential, since the distribution, gathering, and parallel computation of data are concealed within constructors of data types or definitions of parallel skeletons,
3. It provides various kinds of data types such as lists (distributed one-dimensional arrays), matrices (distributed two-dimensional arrays), and trees (distributed binary trees).

SkeTo is implemented in C++ with the MPICH library. A list provided by the current version of SkeTo is restricted to be of fixed length. The most important skeletons for lists are map, reduce, scan, and zip.

The map is a skeleton that applies a function to all elements in a list. The reduce is a skeleton that collapses a list into a single value by repeated applications of a certain associative binary operator. The scan is a skeleton that accumulates all intermediate results for reduce. The zip creates a list of pairs of corresponding elements in two given lists of the same length. By letting $\oplus$ be an associative operator, these four skeletons can be informally defined as

$$\begin{aligned}
\mathsf{map}\ (f,\ [x_1, x_2, \ldots, x_n]) &= [f(x_1),\ f(x_2), \ldots,\ f(x_n)],\\
\mathsf{reduce}\ ((\oplus),\ [x_1, x_2, \ldots, x_n]) &= x_1 \oplus x_2 \oplus \cdots \oplus x_n,\\
\mathsf{scan}\ ((\oplus),\ [x_1, x_2, \ldots, x_n]) &= [x_1,\ x_1 \oplus x_2, \cdots, x_1 \oplus x_2 \oplus \cdots \oplus x_n], \text{and}\\
\mathsf{zip}\ ([x_1, x_2, \ldots, x_n],\ [y_1, y_2, \ldots, y_n]) &= [(x_1,\ y_1),\ (x_2,\ y_2), \ldots,\ (x_n,\ y_n)].
\end{aligned}$$

Table 1. C++ interfaces of existing list.

List skeletons

`dist_list<A>* map(F& f, dist_list<A>* as)`	Map a list
`void map_ow(F& f, dist_list<A>* as)`	Map a list with overwrite
`A reduce(OPLUS& oplus, dist_list<A>* as)`	Reduce a list
`dist_list<A>* scan(OPLUS& oplus, dist_list<A>* as)`	Scan a list
`void scan_ow(OPLUS& oplus, dist_list<A>* as)`	Scan a list with overwrite
`dist_list<std::pair<A,B> >* zip(dist_list<A>* as,dist_list<B>* bs)`	Zip a list

List constructors

`dist_list<A>::dist_list(F& f, int size)`	Create a list using `f`
`dist_list<A>::dist_list(A* array, int size)`	Create a list using `array`

A list is provided as C++ template class `dist_list<A>`, where `A` is a template parameter. This is implemented as a distributed one-dimensional array. A list is initialized by using its constructor in two ways; the first is given the initial data to be distributed to each corresponding node, and the second is given a generator function to compute the initial values of the elements in the list at each node. Table 1 shows the C++ interfaces of the skeletons and constructors of the lists.

In the current implementation of SkeTo, the elements in lists are equally distributed to each node using block placement. The data placement does not change during computation because no skeletons that can dynamically change a list's length are provided.

3 Design of Variable-Length Lists

3.1 Problems That Need Variable-Length Lists

We classified problems that need variable-length lists into three groups.

1. Problems that leave such elements in a given list that satisfy various conditions.
2. Searching problems in which the number of candidates for solutions may dynamically change.
3. Iterative calculations in which computational loads for all elements in a list lack uniformity.

Examples in the first group include the problem of twin primes discussed in Sec. 1 and the problem where the convex hull of a set of points is computed by using the gift-wrapping algorithm. To solve the twin-primes problem, as we have already stated in Sec. 1, we need to add an element to a list, remove an element from it, and concatenate two lists. To solve the problem of the convex hull, we need to extract points that create the convex hull from a list of points and add them into another list. Since fixed-length lists do not have these operations, we cannot solve these problems.

An example in the second group is the Knight's Tour problem. Given a chess board with $N \times M$ squares, this problem involves finding the numbers of paths for a knight that visits each square only once. Even though we can represent possible board states during computation with a fixed-length list, we cannot represent new board states after the next move by the knight. This is because each state may generate zero or many next states and therefore the length of the list has to be dynamically changed. To describe a program for this problem, we have to create a list of possible board states after the first k moves by the knight locally on the master node and then distribute these among nodes for further sequential computation on each node. It is a difficult task to create such a program by only using lists of fixed length.

Examples in the third group include calculations of Mandelbrot and Julia sets, in which the computational load at each point on a plane is dramatically different from the other points. When we represent a set of points on a plane with a fixed-length list, we cannot solve this problem efficiently due to load imbalance. If we can remove elements that have already finished their calculations in a list in iterative calculations (e.g., at regular intervals), we can solve these problems efficiently. Unfortunately, we cannot remove elements from a fixed-length list.

3.2 Skeletons and Operations for Variable-Length Lists

To solve the problems in the previous section, we need operations that add elements to a list, remove elements from it, and concatenate two lists to constitute a long list. To achieve these goals, we propose two skeletons, namely, concatmap and filter, and five operations, namely, append, pushfront, pushback, popfront, and popback. In addition, we revised the definition of zip to accept two lists of different lengths.

The concatmap is a skeleton that applies a function to every element in a list and concatenates the resulting lists to generate a flattened list. The filter is a skeleton that takes a Boolean function, p, and a list, and leaves elements in the list that satisfy p. The append is an operator that concatenates two lists. Note that they do not necessarily have the same length. The popfront and popback are respective operations that remove an element from the front and the back in a list. The pushfront and pushback are respective operations that add an element to the front or the back in a list. They are informally defined as

$$\begin{aligned}
\textsf{concatmap}\ (f,\ [x_1, x_2, \ldots, x_n]) &= [x_{11}, x_{12}, \ldots, x_{1m_1}, x_{21}, x_{22}, \ldots, x_{2m_2}, \\
&\qquad \ldots, x_{n1}, x_{n2}, \ldots, x_{nm_n}], \\
&\quad \textsf{where}\ f\ (x_i) = [x_{i1}, x_{i2}, \ldots, x_{im_i}], \\
\textsf{filter}\ (p,\ [x_1, x_2, \ldots, x_n]) &= \textsf{concatmap}\ (f,\ [x_1, x_2, \ldots, x_n]), \\
&\quad \textsf{where}\ f\ (x) = \text{if}\ p\ (x)\ \text{then}\ [x]\ \text{else}\ [\,], \\
\textsf{append}\ ([x_1, \ldots, x_n],\ [y_1, \ldots, y_m]) &= [x_1, \ldots, x_n, y_1, \ldots, y_m], \\
\textsf{popfront}\ ([x_1, x_2, \ldots, x_n]) &= (x_1,\ [x_2, x_3, \ldots, x_n]), \\
\textsf{popback}\ ([x_1, x_2, \ldots, x_n]) &= (x_n,\ [x_1, x_2, \ldots, x_{n-1}]), \\
\textsf{pushfront}\ (v,\ [x_1, x_2, \ldots, x_n]) &= [v, x_1, x_2, \ldots, x_n],\ \text{and} \\
\textsf{pushback}\ (v,\ [x_1, x_2, \ldots, x_n]) &= [x_1, x_2, \ldots, x_n, v].
\end{aligned}$$

By using these skeletons and operations, we can describe programs for the problems discussed in Sec. 3.1 with variable-length lists. As examples, we have given pseudo-codes for the Knight's Tour problem (Fig. 2) and the Mandelbrot set calculation (Fig. 3).

```
boards = [];   //list of solution boards
// add initial board to boards
pushback(initBoard, boards); // increase boards dynamically up to MAXSIZE
while( length(boards) < MAXSIZE ){
    concatmap(NextBoard(), boards); // generate next moves from each boards
}
// search all solutions with depth first order
concatmap(Solve(), boards); // boards is a list of solution boards
```

Fig. 2. Pseudo-code for Knight's Tour problem

```
points = [...];     // list of points
result_points = []; // list of calculation results
for ( int i=0; i<maxForCount; i++ ){
    map(Calc(), points);// progress calculations in small amounts
    // remove elements that have already finished calculation using filter
    end_points = filter(IsEnd(), points);
    append(result_points, end_points); // add them to result_points
}
append(result_points, points); // result_points is a list of calculation results
```

Fig. 3. Pseudo-code for Mandelbrot-set calculation

4 Implementation

4.1 C++ Interfaces and Program Example

We implemented the variable-length lists as a new library of SkeTo. Variable-length lists are provided as a C++ template class called `dist_list<A>`, which has all interfaces of existing fixed-length lists for compatibility. Thus, a fixed-length list is a special case of a variable-length list where the length of the list never changes. Users do not need to change their existing programs based on fixed-length lists to use the new library. Table 2 shows C++ interfaces of skeletons and operations that were introduced in this new library. Note that `push_front`, `push_back`, `pop_front`, and `pop_back` destructively update the input list as a side effect.

Figure 4 shows a concrete C++ program for the twin primes problem. First, we remove every element whose value is not a prime number with `filter_ow` to create a list of prime numbers. Second, to create a list of pairs of adjacent prime numbers, we apply `zip` to two lists; the first is a list of prime numbers and the second is also a list of prime numbers in which the first prime, 2, is removed by `pop_front`. Finally, we remove every pair whose difference is not two by using `filter_ow` to obtain a list of twin primes.

4.2 Data Structures

There are two requirements for the data structures of variable-length lists. First, each node has to know the latest information on the numbers of elements in other nodes to enable programs to access arbitrary elements in the list and to detect load imbalance. Second, we have to reduce the number of times of data relocation as few as possible because this involves enormous overheads in transferring large amounts of data from one node to another.

Table 2. C++ interfaces of the skeletons and operations

List skeletons

`dist_list<A>* filter(F& f, dist_list<A>* as)`	Filter a list
`void filter_ow(F& f, dist_list<A>* as)`	Filter a list with overwrite
`dist_list<A>* concatmap(F& f, dist_list<A>* as)`	Concatmap a list
`void concatmap_ow(F& f, dist_list<A>* as)`	Concatmap a list with overwrite

List operations

`void dist_list<A>::append(dist_list<A>* as)`	Concatenate lists
`void dist_list<A>::push_front(A v)`	Add `v` at front of a list
`void dist_list<A>::push_back(A v)`	Add `v` to the back of a list
`A dist_list<A>::pop_front()`	Obtain an element from the front of a list
`A dist_list<A>::pop_back()`	Obtain an element from the back of a list

To achieve this end, we introduced *size tables* for the first requirement, and *block-cyclic* placement for the second. A size table has information on the number of elements at each node. When the number of elements changes due to concatmap or filter, the size table in each node is updated using inter-node communication. Moreover, when we concatenate two lists with append, we adopt block-cyclic placement without having to relocate elements on the lists.

Figure 5 shows an example of the initial data structure of a variable-length list, `as`, in which elements in the list have been equally distributed among nodes using block placement. When we remove every element whose value is a multiple of three with `filter_ow`, the number of elements in `as` changes in each node. Thus, the size table at each node is accordingly updated. If the numbers of elements in nodes become too unbalanced, the data in the list are automatically relocated. Returning to the example of

```
dist_list<int>* numbers = new dist_list<int>(SIZE);
dist_list<int>* primes = new dist_list<int>(0);
dist_list<pair<int, int> >* twin_primes;
...add integers, which is more than 1, to numbers...
// make a list of primes using the Sieve of Eratosthenes
int prime;
do {
    prime = numbers->pop_front();
    skeletons::filter_ow(IsNotMultipleOf(prime), numbers);
    primes->push_back(prime);
} while ( val <= sqrt_size );
primes->append(numbers);
// make a list of twin primes from a list of primes
dist_list<int>* dup_primes = primes->clone<int>();
dup_primes->pop_front();
twin_primes = skeletons::zip(primes, dup_primes);
filter_ow(twin_prime, IsCouple());
// twin_primes is solution list
```

Fig. 4. C++ program for twin primes problem

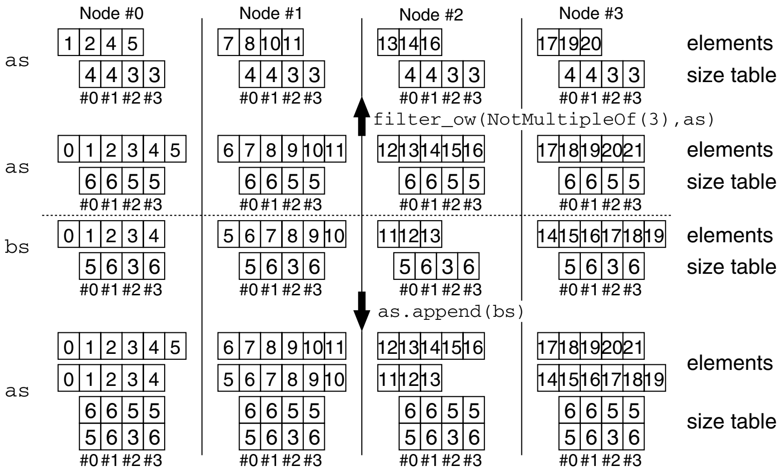

Fig. 5. Data structure of proposed variable-length list

list as, when we concatenate as and bs to make a long list with append, we adopt block-cyclic placement without relocating the entire amount of data in the resulting list. After this, we will call part of a list a *block*, whose elements are sequentially allocated on each node, and will call the number of blocks a *block count*. For example, in Fig. 5, we concatenated two lists (as and bs) whose block counts are both one, and the block count of the resulting list (destructively updated as) is two.

4.3 Implementation of Skeletons

We implemented all skeletons based on the data structures described in the previous subsection. The map, concatmap, and filter skeletons simply apply a given function to each block in the local lists at each node. Note that they do not need any inter-node communication. These skeletons do not degrade their performances due to the number of block counts on a list.

The reduce and scan skeletons need to transfer the results of local calculation from one node to another. The reduce calculates local reductions at each node and then folds all local results by using inter-node communication based on the binary-tree structure of the nodes to obtain the final result. The scan first applies a local scan to each block at each node and then shares the results by using all-to-all communication. Finally, each node calculates the residual local scan. The overheads for these two skeletons due to inter-node communication increase in proportion to the block counts.

The zip skeleton first relocates one of the given lists to make the shapes of the lists the same. It then creates a list of pairs of corresponding elements.

4.4 Data Relocation

In two cases, we have to relocate the data in a list to maintain efficient calculations.

1. When the numbers of elements on each node become too unbalanced.
2. When the block counts of a list reach a large value.

Case 1 is caused by the applications of concatmap or filter. Case 2 is caused by the concatenation of lists with append. This reduces the efficiencies of not only reduce and scan, but also that of updating the size table.

To detect case 1, we introduce the measure $imbalance$ of a list. When the value of $imbalance$ becomes greater than a threshold value, t_u, we relocate the data in the list. The $imbalance$ of list as is defined as:

$$imbalance\,(as) = n \times max\,(P_1, \ldots, P_n)/(P_1 + \cdots + P_n)$$
$$\text{where } P_i = \sum_{j=1}^{m} p_{ij},\ n = \text{the number of nodes},\ m = \text{the block count of } as,$$
$$p_{ij} = \text{the number of elements of the } j\text{-th block at the } i\text{-th node.}$$

Using a rough estimate by letting the value of $imbalance$ be r, it takes r times longer to apply a skeleton than where the numbers of elements in each node are completely even. The value of t_u is empirically determined to be 1.5.

For case 2, we relocate the data in a list when its block count becomes greater than a threshold value, t_b. The value of t_b is empirically determined to be 2048.

The relocation for both cases is performed, if necessary, before each skeleton is applied. After relocation, the elements in the list are evenly distributed with block placement, i.e., the block count becomes 1.

5 Experiment

This section describes the effectiveness of the variable-length lists we propose by presenting the results for micro- and macro-benchmarks. The experimental environment we used was a PC cluster system with 16 processors connected through a 1000 BaseT Ethernet. Each PC had an Intel Pentium 4 (3.0 GHz) CPU and 1 GB of main memory. The operating system was Linux kernel 2.6.8., the compiler was GCC 3.3.5 with optimizing level O2, and the MPI implementation was MPICH 1.2.6.

5.1 Micro-benchmark

The efficiency of skeletons and data relocation degrades as the block count of a list increases. To evaluate these overheads, we measured the execution times for applying map, reduce, and scan to an input list of 80,000,000 elements, increasing the block count of the list from 1 to 4000. We used two functions: the first had a short execution time (only a single multiplication) and the second had a long execution time (1000 multiplications). We also measured the running times for the data relocation in a list.

The results are listed in Table 3. Execution times for block count 1 can be regarded as those of the existing SkeTo implementation that supports only fixed-length lists. Thus, each difference between execution times for block count m ($m > 1$) and 1 is the overhead due to the block-cyclic implementation of variable-length lists. Little overhead was observed in the applications of map to a list even if its block count was large. When the applied function needed a short execution time, we can see rather large overheads in reduce and scan to lists with large block counts. These overheads are due to the cost of inter-node communications for transferring the results of local calculations, and, particularly for scan, the cost of the residual local scan. In contrast, the overhead of these skeletons is relatively small when the applied function needs a long time for execution. The overhead for data relocation is large. Thus, it would be effective to delay the relocation of data.

Table 3. Results for micro-benchmark

	Execution time (s)						
Block count	1	10	100	1000	2000	3000	4000
Map (short duration)	0.0128	0.0129	0.0128	0.0128	0.0129	0.0130	0.0132
Reduce (short duration)	0.0183	0.0182	0.0183	0.0191	0.0194	0.0197	0.0200
Scan (short duration)	0.0407	0.0408	0.0411	0.0443	0.0484	0.0530	0.0580
Map (long duration)	16.8	16.8	16.8	16.8	16.8	16.8	16.8
Reduce (long duration)	16.9	16.9	16.9	16.9	16.9	16.9	17.0
Scan (long duration)	33.8	33.8	33.8	33.9	33.9	34.0	34.1
Data relocation	–	3.74	4.64	4.67	4.66	4.62	4.72

Table 4. Results for macro-benchmark

	Execution time (s) / Ratio				
Number of nodes	1	2	4	8	16
Twin primes	16.86 / 1.00	8.84 / 0.52	4.52 / 0.27	2.38 / 0.14	1.52 / 0.09
Gift-wrapping method	8.59 / 1.00	4.38 / 0.51	2.21 / 0.26	1.13 / 0.13	0.59 / 0.07
Knight's tour	11.92 / 1.00	6.05 / 0.51	3.34 / 0.28	2.53 / 0.21	1.22 / 0.10
Mandelbrot set (var. list)	63.4 / 1.00	31.8 / 0.50	16.2 / 0.26	8.2 / 0.13	4.2 / 0.07
Mandelbrot set (fix. list)	61.2 / 1.00	30.6 / 0.50	29.9 / 0.49	20.8 / 0.34	11.9 / 0.19
Julia set (var. list)	60.0 / 1.00	30.1 / 0.50	15.1 / 0.25	7.6 / 0.13	4.1 / 0.07
Julia set (fix. list)	56.7 / 1.00	28.3 / 0.50	21.6 / 0.38	12.3 / 0.22	6.7 / 0.12

5.2 Macro-benchmark

We measured the execution times of the programs to solve the problems discussed in Sec. 3. We provided the following input to each problem: a list of 10,000,000 integers for the twin primes problem, 1,000,000 randomly generated and uniformly distributed points for the convex hull problem, a 5 $\times$ 6 board for the Knight's Tour, and 1,000 $\times$ 1,000 coordinates for both the Mandelbrot and Julia set problems with 100 iterative calculations $\times$ 100 times. We also measured the execution times of the programs for the Mandelbrot and Julia set problems using fixed-length lists with 10,000 iterations. The results are shown in Table 4. These results indicate excellent performance in all problems with variable-length lists. The programs for the Mandelbrot and Julia set problems with variable-length lists demonstrated particularly good speedups compared to those with fixed-length lists because there was adequate load balancing.

6 Related Work

Skeletal parallel programming was first proposed by Cole [8] and a number of systems (libraries) have been proposed so far. P3L [3] supports both data parallel and task parallel skeletons. A P3L program has a two-layers structure. Higher skeleton level is written in a functional notation, while lower base language level is described in the C language. Muesli [13] is a C++ library that also supports data parallel and task parallel skeletons without syntactic enhancements. Both P3L and Muesli offer lists (distributed one-dimensional arrays) and matrices (distributed two-dimensional arrays). Quaff [11] is another skeleton library in C++. It relies on template-based meta-programming techniques to attain high efficiency. However, these three libraries do not support variable-length lists. The latest version of eSkel [5,9] supports task parallel skeletons for pipelining or master-worker computations, putting emphasis on addressing the issues of nesting of skeletons and interaction between parallel activities. However, it does not support data parallel skeletons for lists like map and reduce. Lithium [1] is a library written in Java that supports common data and task parallel skeletons. Muskel [2], which is a successor of Lithium, is a full Java library targeting workstation clusters, networks, and grids. They are implemented based on (macro) data flow technology rather than template technology exploited by SkeTo. Calcium [6] is also a Java skeleton library on a grid environment. It mainly focuses on performance tuning of skeletal programs.

Another group of libraries that support distributed arrays contains MCSTL[17], and STAPL[18], each of which is an extension of C++ standard template library (STL) for parallel environments. MCSTL has a distributed fixed array whose target is shared-memory multiprocessor environments. STAPL has `pVector` and `pList`, which correspond to variable-length arrays and lists, and it targets both shared- and distributed-memory environments. However, STAPL does not have operations such as zip and concatmap, since they only provide the same operations as STL. Data Parallel Haskell [7,12] offers distributed nested lists in which we can apply filter and concatmap to lists and concatenate lists as well as our variable-length lists. However, it only targets shared-memory multiprocessor environments.

7 Conclusion

We proposed parallel skeletons for variable-length lists and their implementation within a parallel skeleton library called SkeTo. A variable-length list enables us to dynamically increase/decrease the number of elements and thus solve a wide range of problems including those of twin primes, Knight's tour, and Mandelbrot set calculation. Our implementation adopted a block-cyclic representation of lists with size tables, whose efficiency was proved through tests conducted in various experiments. We intend to include the variable-length lists and related skeletons presented in this paper in future releases of SkeTo.

Acknowledgments. We wish to thank Masato Takeichi, Kenetsu Hanabusa, Zhenjiang Hu, Kiminori Matsuzaki, and other POP members in Tokyo for their fruitful discussions on the SkeTo library. This work was partially supported by Grant-in-Aid for Scientific Research (20500029) from the Japan Society of the Promotion of Science.

References

1. Aldinucci, M., Danelutto, M., Teti, P.: An Advanced Environment Supporting Structured Parallel Programming in Java. Future Gener. Comput. Syst. 19(5), 611–626 (2003)
2. Aldinucci, M., Danelutto, M., Dazzi, P.: Muskel: an Expandable Skeleton Environment. Scalable Computing: Practice and Experience 8(4), 325–341 (2007)

3. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P3L: A Structured High Level Programming Language, and its Structured Support. Concurrency: Pract. Exper. 7(3), 225–255 (1995)
4. Backhouse, R.: An Exploration of the Bird-Meertens Formalism. In: STOP Summer School on Constructive Algorithmics, Abeland (1989)
5. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Flexible skeletal programming with eSkel. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 761–770. Springer, Heidelberg (2005)
6. Caromel, D., Leyton, M.: Fine tuning algorithmic skeletons. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 72–81. Springer, Heidelberg (2007)
7. Chakravarty, M.M.T., Leshchinskiy, R., Jones, S.L.P., Keller, G., Marlow, S.: Data Parallel Haskell: A Status Report. In: 2007 ACM Workshop on Declarative Aspects of Multicore Programming (DAMP 2007), pp, pp. 10–18. ACM Press, New York (2007)
8. Cole, M.: Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing, Pitman (1989)
9. Cole, M.: Bringing Skeletons out of the Closet: a Pragmatic Manifesto for Skeletal Parallel Programming. Parallel Comput. 30(3), 389–406 (2004)
10. Darlington, J., Field, A.J., Harrison, P.G., Kelly, P.H.J., Sharp, D.W.N., Wu, Q., While, R.L.: Parallel Programming Using Skeleton Functions. In: Reeve, M., Bode, A., Wolf, G. (eds.) PARLE 1993. LNCS, vol. 694, pp. 146–160. Springer, Heidelberg (1993)
11. Falcou, J., Sérot, J., Chateau, T., Lapresté, J.T.: QUAFF: Efficient C++ Design for Parallel Skeletons. Parallel Comput. 32(7), 604–615 (2006)
12. Jones, S.L.P., Leshchinskiy, R., Keller, G., Chakravarty, M.M.T.: Harnessing the Multicores: Nested Data Parallelism in Haskell. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008 (2008)
13. Kuchen, H.: A skeleton library. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 620–629. Springer, Heidelberg (2002)
14. Matsuzaki, K., Emoto, K., Iwasaki, H., Hu, Z.: A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In: 1st International Conference on Scalable Information Systems, InfoScale 2006 (2006)
15. Rabhi, F.A., Gorlatch, S. (eds.): Patterns and Skeletons for Parallel and Distributed Computing. Springer, Heidelberg (2002)
16. SkeTo Project, `http://www.ipl.t.u-tokyo.ac.jp/sketo/`
17. Singler, J., Sanders, P., Putze, F.: MCSTL: The multi-core standard template library. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 682–694. Springer, Heidelberg (2007)
18. Tanase, G., Bianco, M., Amato, N.M., Rauchwerger, L.: The STAPL pArray. In: 2007 Workshop on Memory performance: Dealing with Applications, systems and architecture (MEDEA 2007), pp. 73–80. ACM Press, New York (2007)