GENERIC PARALLEL PATTERNS IN LIME*

Emil Sekerinski[1] and Tian Zhang[2]

emil@mcmaster.ca, tianz@ca.ibm.com
[1]Department of Computing and Software, McMaster University
[2]IBM Canada Research and Development Centre

November 3, 2014

Lime is a heterogeneous computing language currently being developed at IBM Research[1] [Auerbach et al., 2010]. It is designed to be executable across a broad range of architectures, including CPUs, GPUs, and FPGAs. The novel features of Lime include limiting side-effects and integration of the streaming paradigm into an object-oriented language. Lime extends Java 1.6 as specified by [Gosling et al., 2005].

A design pattern is a reusable solution for a category of problems [Gamma et al., 1995]. Design patterns offer software developers a repertoire of proven solutions that can be tailored for needs. They are highly popular for object-oriented programming and are being used for concurrent and parallel programming. In this document we present a Lime generic implementation of nine parallel design patterns[2]. The first seven patterns, namely fork-join, parallel generator, map, recurrence, stencil, reduction and scan, are taken from [McCool et al., 2012] except parallel generator (which is a variation of fork-join). The last two patterns, namely "divide and conquer differently" and "divide and conquer", the latter one being a special case of the former one, are extracted from various applications.

We divide the package into three parts: "common", which defines patterns and operations that are used by other packages; "testcases", which includes simple examples that can be used for testing the Lime environment; "examples", which consists of practical problems that can be solved using the generic patterns.

First of all, we give the list of patterns here, the details of which will be revealed later:

<src/common/GenericPatterns.lime 1>

```
1  package common;
2
3  public class GenericPatterns {
4
5      /* generic type names:
6       * TI: input type
7       * TO: output type
8       * T: when input and output types are the same
9       * TN: numeric type with ''+'' operation, which is necessary for scan pattern */
10
11     <<Fork-join Pattern 9>>
12
13     <<Parallel Generator Pattern 11>>
14
15     <<Map Pattern 13>>
16
17     <<Recurrence Pattern 15>>
18
19     <<Stencil Pattern 17>>
```

*This document is prepared using Literate Programming for Eclipse: http://lep.sourceforge.net/.

[1]Lime is currently in alpha stage, installation guide available at http://lime.watson.ibm.com/wiki/articles/t/h/e/The_Liquid_Metal_Alpha_Release.html

[2]Some patterns (e.g., naive map pattern) might be less efficient than Lime native implementation.

```
20
21      <<Reduction Pattern 19>>
22
23      <<Scan Pattern 21>>
24
25      <<Divide and Conquer Differently Pattern 23>>
26
27      <<Divide and Conquer Pattern 25>>
28
29  }
```

INCLUDED BLOCKS: 9 on page 9, 11 on page 10, 13 on page 11, 15 on page 13, 17 on page 14, 19 on page 15, 21 on page 16, 23 on page 18, 25 on page 20

Then we define some constants and operations that will be used in examples:

<src/common/Toolkit.lime 2>

```
1   package common;
2
3   import lime.util.PseudoRandom;
4
5   public class Toolkit {
6
7       /* maximum random integer */
8       public static final int MAX = 15;
9
10      /* test messages */
11      public static final string INPUT     = "input:     ";
12      public static final string RESULT    = "result:    ";
13      public static final string REFERENCE = "reference: ";
14      public static final string TEST_PASS = "TEST PASSED!";
15      public static final string TEST_FAIL = "TEST FAILED!";
16
17      /* increases an integer by 1 */
18      public static local int inc(int i) {
19          return i + 1;
20      }
21
22      /* generate 2 neighbors on a circular array */
23      public static local <NN extends int> `(int, int)[[NN]] genNeighbors(int[[NN]] a) {
24          `(int, int)[NN] result = new `(int, int)[NN];
25          for (NN i) {
26              result[i] = `(a[(NN) (i - 1)], a[(NN) (i + 1)]);
27          }
28          return new `(int, int)[[NN]](result);
29      }
30
31      /* addition of 2 integers */
32      public static local int add(`(int, int) a) {
33          return a.0 + a.1;
34      }
35
36      /* multiplication of 2 integers  */
37      public static local int mult(int[[2]] a) {
38          return a[0] * a[1];
39      }
40
41      /* determines if integer p is a power of integer n */
42      public static boolean isPowerOf(int p, int i) {
```

```
43          assert i >= 2;
44          while (p >= i && p % i == 0) {
45              p = p / i;
46          }
47          return (p == 1);
48      }
49
50      /* outputs a array of type T and length NN */
51      public static <T extends Value, NN extends int> void output(T[NN] a) {
52          System.out.println(a.contentToString());
53      }
54
55      /* outputs a value array of type T and length NN */
56      public static <T extends Value, NN extends int> void output(T[[NN]] array) {
57          System.out.println(array.contentToString());
58      }
59
60      /* generates a pseudo-random integer value array of length NN */
61      public glocal static <NN extends int> int[[NN]] randomIntArray(long seed) {
62          PseudoRandom rand = new PseudoRandom();
63          rand.setSeed(seed);
64          int[NN] result = new int[NN];
65          for (NN i) {
66              result[i] = rand.nextInt(Toolkit.MAX);
67          }
68          return new int[[NN]](result);
69      }
70
71      /* generates a pseudo-random complex value array of length NN */
72      public glocal static <NN extends int> complex[[NN]] randomComplexArray(long seed) {
73          PseudoRandom rand = new PseudoRandom();
74          rand.setSeed(seed);
75          complex[NN] result = new complex[NN];
76          for (NN i) {
77              result[i] = new complex((float) rand.nextDouble() * 1e-6f, (float) rand.
                    nextDouble() * 1e-6f);
78          }
79          return new complex[[NN]](result);
80      }
81
82  }
```

We also need a class as the sink of value array:

<src/common/ValueArraySink.lime 3>

```
1   package common;
2
3   public class ValueArraySink<T extends Value, NN extends int> {
4
5       private int index;
6       private T[NN] result;
7
8       public ValueArraySink() {
9           this.index = 0;
10          this.result = new T[NN];
11      }
12
13      /* records a value */
14      public void dump(T t) {
```

```
15          result[(NN) index++] = t;
16      }
17
18      /* returns the result as a value array */
19      public T[[NN]] getResult() {
20          return new T[[NN]](result);
21      }
22
23      /* compares result with reference and outputs the result */
24      public void compareWith(T[NN] reference) {
25          System.out.print(Toolkit.RESULT);
26          Toolkit.output(result);
27          System.out.print(Toolkit.REFERENCE);
28          Toolkit.output(reference);
29          if (new T[[NN]](result) == new T[[NN]](reference)) {
30              System.out.println(Toolkit.TEST_PASS);
31          } else {
32              System.err.println(Toolkit.TEST_FAIL);
33          }
34      }
35
36  }
```

as well as a sink of a single value:

<src/common/ValueSink.lime 4>

```
1   package common;
2
3   public class ValueSink<T extends Value> {
4
5       private T result;
6
7       /* records the value */
8       public void dump(T t) {
9           result = t;
10      }
11
12      /* returns the result */
13      public T getResult() {
14          return result;
15      }
16
17      /* compares result with reference and outputs the result */
18      public void compareWith(T reference) {
19          System.out.println(Toolkit.RESULT + result);
20          System.out.println(Toolkit.REFERENCE + reference);
21          if (result == reference) {
22              System.out.println(Toolkit.TEST_PASS);
23          } else {
24              System.err.println(Toolkit.TEST_FAIL);
25          }
26      }
27
28  }
```

Here is a value array generator, that generates a value at a time from a given value array:

<src/common/ValueArrayGenerator.lime 5>

```
package common;

public final class ValueArrayGenerator<T extends Value, NN extends int> {

    private int index;
    private T[[NN]] array;

    /* makes a source task from the given value array */
    public static task IsoTask<Void, T> makeSource(T[[NN]] array) {
        return task (new ValueArrayGenerator<T, NN>(array)).next;
    }

    /* initializes the value array */
    public local ValueArrayGenerator(T[[NN]] array) {
        this.index = 0;
        this.array = array;
    }

    /* generates the next value */
    public final local T next() {
        if (index < NN.size) {
            return array[(NN) index++];
        } else {
            throw new StreamUnderflow();
        }
    }

}
```

A timer for performance measurements:

<src/common/Timer.lime 6>

```
package common;

public class Timer {

    long timer, startTime;

    public void start() {
        timer = 0;
        startTime = System.nanoTime();
    }

    public void stop() {
        timer += System.nanoTime() - startTime;
        System.out.println("Time: " + (timer / 1e9) + "s");
    }

}
```

Here are some commonly used permutations in stream computing:

<src/common/Permutations.lime 7>

```
package common;

public class Permutations {

    /* identity on array */
    public static local <T extends Value, NN extends int> T[[NN]] id(T[[NN]] array) {
```

```
7           return array;
8       }
9
10      /* wraps up an IsoTask<TI[[NN]], TO[[NN]]> task to be IsoTask<TI, TO> */
11      public task static local <TI extends Value, TO extends Value, NN extends int> IsoTask<TI
            , TO> serialize(IsoTask<TI[[NN]], TO[[NN]]> F) {
12          return (TI # TI[[NN]])
13              => F
14              => (TO[[NN]] # TO);
15      }
16
17      /* wraps up an IsoTask<TI, TO> task to be IsoTask<TI[[NN]], TO[[NN]]> */
18      public task static local <TI extends Value, TO extends Value, NN extends int> IsoTask<TI
            [[NN]], TO[[NN]]> parallelize(IsoTask<TI, TO> F) {
19          return (TI[[NN]] # TI)
20              => F
21              => (TO # TO[[NN]]);
22      }
23
24      /* reorders the array before splitting */
25      public static local <T extends Value, NN extends int> T[[NN]] split_even_odd(T[[NN]]
            input) {
26          T[NN] result = new T[NN](input);
27          final int H = NN.size / 2;
28          for (int<H> i) {
29              result[(NN) (2 * i)] = input[i];
30              result[(NN) (2 * i + 1)] = input[(NN) (i + H)];
31          }
32          return new T[[NN]](result);
33      }
34
35      /* restores the array after splitting */
36      public static local <T extends Value, NN extends int> T[[NN]] join_even_odd(T[[NN]]
            input) {
37          T[NN] result = new T[NN](input);
38          final int H = NN.size / 2;
39          for (int<H> i) {
40              result[i] = input[(NN) (2 * i)];
41              result[(NN) (i + H)] = input[(NN) (2 * i + 1)];
42          }
43          return new T[[NN]](result);
44      }
45
46      /* reverse the bits of each index */
47      public static local <T extends Value, NN extends int> T[[NN]] bit_reverse(T[[NN]] input)
            {
48          T[NN] result = new T[NN](input);
49          final int N = NN.size;
50          for (NN i = 0, j = 0; i < N - 1; i++) {
51              int k = N / 2;
52              if (i < j) {
53                  T tmp = result[i];
54                  result[i] = result[j];
55                  result[j] = tmp;
56              }
57              while (k <= j) {
58                  j = (NN) (j - k);
59                  k = k / 2;
60              }
```

```
                j += k;
            }
            return new T[[NN]](result);
        }

}
```

A value class for complex numbers (taken from Lime sample project):

<src/common/complex.lime 8>

```
package common;

/**
 * A single precision complex number
 * Taken from Lime sample project (20141016 build)
 */
public final value class complex {

    public final double real;
    public final double imag;

    public static final complex ZERO = new complex(0.0f, 0.0f);
    public static final complex ONE  = new complex(1.0f, 0.0f);

    public complex(double real, double imag) {
        this.real = real;
        this.imag = imag;
    }

    public complex(double real) {
        this(real, 0.0f);
    }

    public complex - this {
        return new complex(-real, -imag);
    }

    public complex + this {
        return this;
    }

    public complex conjugate() { // TODO: could use "~" prefix operator.  good idea or not?
        return new complex(real, -imag);
    }

    public complex this + (complex that) {
        return new complex(real+that.real, imag+that.imag);
    }

    public complex this - (complex that) {
        return this + -that;
    }

    public complex this * (complex that) {
        double r = real*that.real - imag*that.imag;
        double i = real*that.imag + imag*that.real;
        return new complex(r, i);
    }

```

```
50      public complex this * (double that) {
51          return new complex(real * that, imag * that);
52      }
53
54      public complex this / (complex that) {
55  //      if (that == ZERO) {
56  //          throw new ArithmeticException("Complex / by zero");
57  //      }
58
59          complex conj = that.conjugate();
60          complex numerator = this * conj;
61          double denominator = (that * conj).real; // imag should be 0
62
63          return new complex(numerator.real/denominator, numerator.imag/denominator);
64      }
65
66      public complex +++this {
67          return this + ONE;
68      }
69
70      public complex ---this {
71          return this - ONE;
72      }
73
74      public boolean this < (complex that) {
75          if (real == that.real)
76              return imag < that.imag;
77          else
78              return real < that.real;
79      }
80
81      public boolean this <= (complex that) {
82          return this == that || this < that;
83      }
84
85      public boolean this > (complex that) {
86          return that < this;
87      }
88
89      public boolean this >= (complex that) {
90          return that <= this;
91      }
92
93      public int compareTo(complex that) {
94          return this == that ? 0 : this < that ? -1 : 1;
95      }
96
97      public boolean isReal() {
98          return imag == 0.0;
99      }
100
101     public string tostring() {
102         if (imag == 0.0) {
103             return ""+real;
104         } else if (real == 0.0) {
105             return ""+imag+"i";
106         } else {
107             return "" + real + (imag >= 0.0 ? "+" : "") + imag + "i";
108         }
```

```
109     }
110
111 }
```

The fork-join pattern lets control flow fork into multiple parallel flows that rejoin later [McCool et al., 2012]. A typical use is to create a group of tasks, applying the same operation in parallel. In Lime it can easily be implemented by using splitter and joiner:

<Fork-join Pattern 9>

```
1     /* fork-join pattern */
2
3     /* PP: the number of parallel subtasks
4      * F: the task to be parallelized */
5     public task static local <TI extends Value, TO extends Value, PP extends int> IsoTask<TI
          , TO> forkjoin(IsoTask<TI, TO> F) {
6         final int P = PP.size;
7         return (TI # TI[[P]])
8             // TI[[P]]
9             => task split TI[[P]]
10            // '(TI, TI, ..., TI)
11            => task [ (P) F ]
12            // '(TO, TO, ..., TO)
13            => task join TO[[P]]
14            // TO[[P]]
15            => (TO[[P]] # TO);
16    }
17
18    /* fork-join pattern that maintains the order */
19
20    /* SS: the size of subtasks
21     * PP: the number of parallel subtasks
22     * F: the task to be parallelized */
23    public task static local <TI extends Value, TO extends Value, SS extends int, PP extends
           int> IsoTask<TI, TO> forkjoinOrdered(IsoTask<TI[[SS]], TO[[SS]]> F) {
24        final int P = PP.size;
25        return (TI # TI[[P][SS]])
26            // TI[[P][SS]]
27            => task split TI[[P][SS]]
28            // '(TI[SS], TI[SS], ..., TI[SS])
29            => task [ (P) F ]
30            // '(TO[SS], TO[SS], ..., TO[SS])
31            => task join TO[[P][SS]]
32            // TO[[P][SS]]
33            => (TO[[P][SS]] # TO);
34    }
```

USED IN: src/common/GenericPatterns.lime on page 1

One test case is to create 4 parallel tasks that all increase an integer by 1:

<src/testcases/forkjoin/Inc.lime 10>

```
1  package testcases.forkjoin;
2
3  import common.*;
4  import lime.util.Tasks;
5
6  public class Inc {
7
```

```
    /* apply task inc to an integer array */
    public static void main(String[] args) {
        System.out.println(Inc.class.getName());
        final int N = 8, P = 4;
        int[[N]] intN = Toolkit.<N>randomIntArray(System.nanoTime());

        // computes the result
        ValueArraySink<int, N> sink = new ValueArraySink<int, N>();
        var t = Tasks.source(intN)
             => ([ GenericPatterns.<int, int, P>forkjoin(task Toolkit.inc) ])
             => task sink.dump;
        Timer timer = new Timer();
        timer.start();
        t.finish();
        timer.stop();

        // computes the reference and compares
        int[N] reference = new int[N](intN);
        for (int<N> i) {
            reference[i] = Toolkit.inc(reference[i]);
        }
        sink.compareWith(reference);
    }

}
```

A variation of fork-join pattern would be parallel generator pattern, which generates values in parallel:

<Parallel Generator Pattern 11>

```
    /* parallel generator pattern */

    /* PP: the number of parallel generators
     * gen: the generator tasks to be parallelized */
    public task static local <T extends Value, PP extends int> IsoTask<Void, T>
        parallelGenerator(IsoTask<Void, T>[[PP]] gens) {
        return task [ gens ]
            // '(T, T, ..., T)
            => task join T[[PP]]
            // T[[P]]
            => (T[[PP]] # T);
    }
```

USED IN: src/common/GenericPatterns.lime on page 1

We use random integer value arrays to test:

<src/testcases/pargen/Weave.lime 12>

```
package testcases.pargen;

import common.*;
import lime.util.Tasks;

public class Weave {

    static final int N = 4, P = 4, TOTAL = N * P;

    /* initializes the generators as repeatable expressions */
    static local <NN extends int> IsoTask<Void, int>[[NN]] init() {
```

```
        final int n = NN.size;
        IsoTask<Void, int>[[1]] t = { ValueArrayGenerator<int, N>.makeSource(Toolkit.<N>
            randomIntArray(n)) };
        if (n == 1) {
            return new IsoTask<Void, int>[[NN]](t);
        } else {
            final int SUB = n - 1;
            return new IsoTask<Void, int>[[NN]](Weave.<SUB>init() + t);
        }
    }

    /* weaves N integer value arrays */
    public static void main(String[] args) {
        System.out.println(Weave.class.getName());
        System.out.print(Toolkit.INPUT);
        int[P][N] intN = new int[P][N];
        for (int<P> i) {
            intN[i] = new int[N](Toolkit.<N>randomIntArray(i + 1));
            Toolkit.output(intN[i]);
        }

        // computes the result
        ValueArraySink<int, TOTAL> sink = new ValueArraySink<int, TOTAL>();
        var t = ([ GenericPatterns.<int, P>parallelGenerator(Weave.<P>init()) ])
             => task sink.dump;
        Timer timer = new Timer();
        timer.start();
        t.finish();
        timer.stop();

        // computes the reference and compares
        int[TOTAL] reference = new int[TOTAL];
        for (int<N> i) {
            for (int<P> j) {
                reference[i * P + j] = intN[j][i];
            }
        }
        sink.compareWith(reference);
    }

}
```

In a map pattern, a function is applied to all elements of a collection, usually producing a new collection with the same shape as the input, and the elementary function must be pure (without side effects) [McCool et al., 2012].

<Map Pattern 13>

```
    /* map pattern */

    /* basic version, identical to input
     * F: the map filter */
    public task static local <TI extends Value, TO extends Value> IsoTask<TI, TO> map(
        IsoTask<TI, TO> F) {
        return F;
    }

    /* parallel version
     * PP: the number of parallel subtasks
     * F: the map filter */
```

```
12      public task static local <TI extends Value, TO extends Value, PP extends int> IsoTask<TI
            , TO> mapParallel(IsoTask<TI, TO> F) {
13          return GenericPatterns.<TI, TO, PP>forkjoin(F);
14      }
15
16      /* array version
17       * NN: input array length
18       * F: the map filter */
19      public task static local <TI extends Value, TO extends Value, NN extends int> IsoTask<TI
            [[NN]], TO[[NN]]> mapArray(IsoTask<TI, TO> F) {
20          return (TI[[NN]] # TI)
21              // TI
22              => F
23              // TO
24              => (TO # TO[[NN]]);
25      }
26
27      /* parallel version on value array
28       * NN: input array length
29       * PP: the number of parallel subtasks
30       * F: the map filter */
31      public task static local <TI extends Value, TO extends Value, NN extends int, PP extends
             int> IsoTask<TI[[NN]], TO[[NN]]> mapArrayParallel(IsoTask<TI, TO> F) {
32          return (TI[[NN]] # TI)
33              // TI
34              => GenericPatterns.<TI, TO, PP>forkjoin(F)
35              // TO
36              => (TO # TO[[NN]]);
37      }
```

USED IN: src/common/GenericPatterns.lime on page 1

The following program tests *MapParallel*:

<src/testcases/map/Inc.lime 14>

```
1   package testcases.map;
2
3   import common.*;
4   import lime.util.Tasks;
5
6   public class Inc {
7
8       /* applys task inc to an integer array */
9       public static void main(String[] args) {
10          System.out.println(Inc.class.getName());
11          final int N = 8, P = 4;
12          int[[N]] intN = Toolkit.<N>randomIntArray(System.nanoTime());
13          System.out.print(Toolkit.INPUT);
14          Toolkit.<int, N>output(intN);
15
16          // computes the result
17          ValueArraySink<int, N> sink = new ValueArraySink<int, N>();
18          var t = Tasks.source(intN)
19              => ([ GenericPatterns.<int, int, P>mapParallel(task Toolkit.inc) ])
20              => task sink.dump;
21          Timer timer = new Timer();
22          timer.start();
23          t.finish();
24          timer.stop();
```

```
25
26          // computes the reference and compares
27          int[N] reference = new int[N](intN);
28          for (int<N> i) {
29              reference[i] = Toolkit.inc(reference[i]);
30          }
31          sink.compareWith(reference);
32      }
33
34  }
```

A recurrence is like a map but where elements can use the outputs of adjacent elements as inputs [McCool et al., 2012]. Here we show how iteration can be done in this style.

<Recurrence Pattern 15>

```
1       /* recurrence pattern
2
3       /* NN: number of iterations
4        * F: the iterative operation */
5       public task static local <T extends Value, NN extends int> IsoTask<T, T> recurrence(
            IsoTask<T, T> F) {
6           if (NN.size > 1) {
7               final int SUB = NN.size - 1;
8               return F
9                   // T
10                  => GenericPatterns.<T, SUB>recurrence(F);
11          }
12          else {
13              return F;
14          }
15      }
```

USED IN: src/common/GenericPatterns.lime on page 1

Here is a test case that increases integer values multiple times:

<src/testcases/recurrence/Inc.lime 16>

```
1   package testcases.recurrence;
2
3   import common.*;
4   import lime.util.Tasks;
5
6   public class Inc {
7
8       /* applys task inc to an integer value array multiple times */
9       public static void main(String[] args) {
10          System.out.println(Inc.class.getName());
11          final int N = 8, R = 4;
12          int[[N]] intN = Toolkit.<N>randomIntArray(System.nanoTime());
13          System.out.print(Toolkit.INPUT);
14          Toolkit.<int, N>output(intN);
15
16          // computes the result
17          ValueArraySink<int, N> sink = new ValueArraySink<int, N>();
18          var t = Tasks.source(intN)
19               => ([ GenericPatterns.<int, R>recurrence(task Toolkit.inc) ])
20               => task sink.dump;
21          Timer timer = new Timer();
```

```
22          timer.start();
23          t.finish();
24          timer.stop();
25
26          // computes the reference and compares
27          int[N] reference = new int[N](intN);
28          for (int<N> i) {
29              for (int<R> j) {
30                  reference[i] = Toolkit.inc(reference[i]);
31              }
32          }
33          sink.compareWith(reference);
34      }
35
36  }
```

The stencil pattern is a generalization of the map pattern where the elemental function can access not only a single element in an input collection but also a set of "neighbors" [McCool et al., 2012].

<Stencil Pattern 17>

```
1       /* stencil pattern */
2
3       /* basic version
4        * CALC: calculates the stream of new values
5        * TI: supposed to be '(TO, TO, ... , TO) */
6       public task static local <TI extends Value, TO extends Value> IsoTask<TI, TO> stencil(
            IsoTask<TI, TO> CALC) {
7           return GenericPatterns.<TI, TO>map(CALC);
8       }
9
10      /* parallel version
11       * CALC: calculates the stream of new values
12       * TI: supposed to be '(TO, TO, ... , TO) */
13      public task static local <TI extends Value, TO extends Value, PP extends int>IsoTask<TI,
             TO> stencilParallel(IsoTask<TI, TO> CALC) {
14          return GenericPatterns.<TI, TO, PP>mapParallel(CALC);
15      }
```

USED IN: src/common/GenericPatterns.lime on page 1

The following test case shows how to use stencil pattern to calculate the sum of the two neighbors of each element in an integer value array:

<src/testcases/stencil/SumNeighbors.lime 18>

```
1   package testcases.stencil;
2
3   import common.*;
4   import lime.util.Tasks;
5
6   public class SumNeighbors {
7
8       /* calculates the sum of the two neighbors of each element in an integer value array */
9       public static void main(String[] args) {
10          System.out.println(SumNeighbors.class.getName());
11          final int N = 8, P = 4;
12          int[[N]] intN = Toolkit.<N>randomIntArray(System.nanoTime());
13          System.out.print(Toolkit.INPUT);
14          Toolkit.<int, N>output(intN);
```

```
15
16          // computes the result
17          ValueArraySink<int, N> sink = new ValueArraySink<int, N>();
18          var t = Tasks.source(intN)
19               => (# int[[8]])
20               => task Toolkit.<8>genNeighbors
21               => (# `(int, int))
22               => ([ GenericPatterns.<`(int, int), int, P>stencilParallel(task Toolkit.add) ])
23               => task sink.dump;
24          Timer timer = new Timer();
25          timer.start();
26          t.finish();
27          timer.stop();
28
29          // computes the reference and compares
30          int[N] reference = new int[N](intN);
31          int s = 0;
32          for (int<N> i) {
33              s = Toolkit.add(`(intN[(int<N>) (i - 1)], intN[(int<N>) (i + 1)]));
34              reference[i] = s;
35          }
36          sink.compareWith(reference);
37      }
38
39  }
```

A reduction combines all the elements in a collection into a single element using an associative combiner function. Because the combiner function is associative, many orderings are possible [McCool et al., 2012].

<Reduction Pattern 19>

```
1       /* reduction pattern */
2
3       /* F: reduction on PP elements
4        * NN: the size of the value array
5          PP: the number of sub-problems in each step */
6       public task static local <T extends Value, NN extends int, PP extends int> IsoTask<T, T>
            reduction(IsoTask<T[[PP]], T> F) {
7           final int N = NN.size;
8           final int P = PP.size;
9           assert N >= P && Toolkit.isPowerOf(N, P);
10          final int SUB = N / P;
11          if (N > P) {
12              IsoTask<T, T> subreduction = GenericPatterns.<T, SUB, PP>reduction(F);
13              return GenericPatterns.<T, T, PP>forkjoin(subreduction)
14                  // T
15                  => (T # T[[PP]])
16                  // T[[PP]]
17                  => F;
18          }
19          else {
20              // apply F when only PP elements
21              return (T # T[[PP]])
22                  => F;
23          }
24      }
```

USED IN: src/common/GenericPatterns.lime on page 1

Here is a test case of computing the product of an integer value array:

<src/testcases/reduction/Product.lime 20>

```
1  package testcases.reduction;
2
3  import common.*;
4  import lime.util.Tasks;
5
6  public class Product {
7
8      /* calculates the product of an integer value array */
9      public static void main(String[] args) {
10         System.out.println(Product.class.getName());
11         final int N = 8;
12         int[[N]] intN = Toolkit.<N>randomIntArray(System.nanoTime());
13         System.out.print(Toolkit.INPUT);
14         Toolkit.<int, N>output(intN);
15
16         // computes the result
17         ValueSink<int> sink = new ValueSink<int>();
18         var t = Tasks.source(intN)
19              => ([ GenericPatterns.<int, N, 2>reduction(task Toolkit.mult) ])
20              => task sink.dump;
21         Timer timer = new Timer();
22         timer.start();
23         t.finish();
24         timer.stop();
25
26         // computes the reference and compares
27         int reference = 1;
28         int[2] a = new int[2];
29         for (int<N> i) {
30             a[0] = reference;
31             a[1] = intN[i];
32             reference = Toolkit.mult(new int[[2]](a));
33         }
34         sink.compareWith(reference);
35     }
36
37 }
```

Scan computes all partial reductions of a collection. In other words, for every output position, a reduction of the input up to that point is computed [McCool et al., 2012]. Note that here the basic operation must be on two elements, otherwise not every element has a partial reduction up to it.

<Scan Pattern 21>

```
1      /* scan pattern */
2
3      /* TN: a generic numeric type with + operation, is necessary for method scan_merge */
4      public task static local <TN extends numeric<TN>, NN extends int> IsoTask<TN[NN], TN[NN]> scan()
           {
5          final int N = NN.size;
6          assert N >= 2 && Toolkit.isPowerOf(N, 2);
7          if (N > 2) {
8              final int H = N / 2;
9              return GenericPatterns.<TN, TN, H, 2>forkjoinOrdered(Permutations.<TN, TN, H>
                   parallelize(GenericPatterns.<TN, H>scan()))
10                 // TN
11                 => Permutations.<TN, TN, NN>serialize(task GenericPatterns.<TN, NN>
                       scan_merge);
```

```
        }
        else {
            return Permutations.<TN, TN, NN>serialize(task GenericPatterns.<TN, NN>
                scan_merge);
        }
    }

    /* merges the two scanned parts */
    public static local <TN extends numeric<TN>, NN extends int> TN[[NN]] scan_merge(TN[[NN
        ]] input) {
        TN[NN] result = new TN[NN](input);
        final int N = NN.size;
        final int H = N / 2;
        // the second half increased by i[m - 1]
        // could not find a equivalent operation on TI with a filter as parameter,
        // so TN was introduced
        for (NN i = (NN) H; i > 0; i++) {
            result[i] += result[(NN) (H - 1)];
        }
        return new TN[[NN]](result);
    }
```

USED IN: src/common/GenericPatterns.lime on page 1

The following test case computes scan of summation:

<src/testcases/scan/Sum.lime 22>

```
package testcases.scan;

import common.*;
import lime.util.Tasks;

public class Sum {

    /* calculates scan of summation */
    public static void main(String[] args) {
        System.out.println(Sum.class.getName());
        final int N = 8;
        int[[N]] intN = Toolkit.<N>randomIntArray(System.nanoTime());
        System.out.print(Toolkit.INPUT);
        Toolkit.<int, N>output(intN);

        // computes the result
        ValueArraySink<int, N> sink = new ValueArraySink<int, N>();
        var t = Tasks.source(intN)
             => ([ GenericPatterns.<int, 8>scan() ])
             => task sink.dump;
        Timer timer = new Timer();
        timer.start();
        t.finish();
        timer.stop();

        // computes the reference and compares
        int[N] reference = new int[N](intN);
        int s = 0;
        for (int<N> i) {
            s = Toolkit.add(`(s, reference[i]));
            reference[i] = s;
        }
```

```
        sink.compareWith(reference);
    }

}
```

Divide and conquer differently pattern divides the problem into several sub-problems that potentially have different solutions, solve them in parallel, and merge the outputs into the final result.

<Divide and Conquer Differently Pattern 23>

```
    /* divide and conquer differently pattern */

    /* NN: the size of the value array
     * SS: the size of sub-problems
     * PP: the number of sub-problems in each step */
    public task static local <T extends Value, NN extends int, SS extends int, PP extends
        int> IsoTask<T, T> dncd(IsoTask<T[[SS]], T[[SS]]>[[PP]] subs, IsoTask<T, T> merge) {
        final int N = NN.size;
        final int P = PP.size;
        final int S = N / P;
        assert N == S * P && P >= 2 && Toolkit.isPowerOf(N, P);
        if (N > P) {
            return (T # T[[PP][SS]])
                // T[[PP][SS]]
                => task split T[[PP][SS]]
                // '(T[SS], T[SS], ..., T[SS])
                => task [ subs ]
                // '(T[SS], T[SS], ..., T[SS])
                => task join T[[PP][SS]]
                // T[[PP][SS]]
                => (T[[PP][SS]] # T)
                // T
                => merge;
        }
        else {
            return merge;
        }
    }

    /* by default PP = 2 */
    public task static local <T extends Value, NN extends int, HH extends int> IsoTask<T, T>
        dncdD(IsoTask<T[[HH]], T[[HH]]>[[2]] subs, IsoTask<T, T> merge) {
        return GenericPatterns.<T, NN, HH, 2>dncd(subs, merge);
    }
```

USED IN: src/common/GenericPatterns.lime on page 1

An application would be bitonic sort (`http://en.wikipedia.org/wiki/Bitonic_sorter`), where the first half and the second half have different values of parameter *upsort* (part of the code is taken from Lime sample projects).

<src/examples/dncd/BitonicSort.lime 24>

```
package examples.dncd;

import common.*;
import lime.util.Tasks;

public class BitonicSort {

    /**
```

```
9         * Compares the two input keys and exchanges their order if they are
10        * not sorted.
11        *
12        * upsort determines if the sort is nondecreasing (UP) or
13        * nonincreasing (DOWN).  'true' indicates UP sort and 'false'
14        * indicates DOWN sort.
15        */
16       static local '(int, int) compareExchange('(int, int) k, boolean upsort) {
17           int mink, maxk;
18
19           if (k.0 <= k.1) {
20               mink = k.0;
21               maxk = k.1;
22           }
23           else { /* k0 > k1 */
24               mink = k.1;
25               maxk = k.0;
26           }
27
28           if (upsort) {
29               return '(mink, maxk); /* UP sort */
30           }
31           else {
32               return '(maxk, mink); /* DOWN sort */
33           }
34
35       }
36
37       /**
38        * Partitions the input bitonic sequence of length L into two bitonic
39        * sequences of length L/2, with all numbers in the first sequence <=
40        * all numbers in the second sequence if sort direction is UP (similar
41        * case for DOWN sortdir)
42        *
43        * Graphically, it is a bunch of CompareExchanges with same sort direction,
44        * clustered together in the sort network at a particular step (of
45        * some merge stage).
46        */
47
48       task static local <L extends int> IsoTask<int, int> bitonic_compare(boolean upsort) {
49           final int H = L.size / 2;
50           var xchange = (int # '(int, int))
51                       => task compareExchange('(int, int), upsort)
52                       => (# int);
53           /* Each compareExchange examines keys that are L/2 elements apart */
54           return (int # int[[H]])
55               => task split int[[H]]
56               => task [ (H) xchange ]
57               => task join int[[H]]
58               => (int[[H]] # int);
59       }
60
61       static local <L extends int> IsoTask<int, int> bitonic_level(boolean upsort) {
62           final int LL = L.size;
63           final int H = L.size / 2;
64           if (LL > 2) {
65               return BitonicSort.<LL>bitonic_compare(upsort)
66                   => BitonicSort.<H>bitonic_level(upsort);
67           } else {
```

```
68              return BitonicSort.<2>bitonic_compare(upsort);
69          }
70      }
71
72      task static local <NN extends int> IsoTask<int, int> construct(boolean upsort) {
73          final int N = NN.size;
74          if (N > 2) {
75              final int H = N / 2;
76              IsoTask<int[[H]], int[[H]]>[2] subs = new IsoTask<int[[H]], int[[H]]>[2];
77              subs[0] = Permutations.<int, int, H>parallelize(BitonicSort.<H>construct(upsort)
                    );
78              subs[1] = Permutations.<int, int, H>parallelize(BitonicSort.<H>construct(!upsort
                    ));
79              IsoTask<int[[H]], int[[H]]>[[2]] subtasks = new IsoTask<int[[H]], int[[H
                    ]]>[[2]](subs);
80              return GenericPatterns.<int, NN, H, 2>dncd(subtasks, BitonicSort.<NN>
                    bitonic_level(upsort));
81          } else {
82              return BitonicSort.<NN>bitonic_level(upsort);
83          }
84      }
85
86      public static void main(String[] args) {
87          final int N = 8;
88          int[[N]] intN = Toolkit.<N>randomIntArray(System.nanoTime());
89          System.out.print(Toolkit.INPUT);
90          Toolkit.<int, N>output(intN);
91
92          // computes the result
93          ValueArraySink<int, N> sink = new ValueArraySink<int, N>();
94          var t = Tasks.source(intN)
95               => ([ BitonicSort.<8>construct(true) ])
96               => task sink.dump;
97          Timer timer = new Timer();
98          timer.start();
99          t.finish();
100         timer.stop();
101
102         // computes the reference and compares
103         int[N] reference = new int[N](intN);
104         reference.sort();
105         sink.compareWith(reference);
106     }
107
108 }
```

Divide and conquer pattern is the special case of divide and conquer differently pattern, where all the sub-solutions are the same.

<Divide and Conquer Pattern 25>

```
1       /* divide and conquer pattern */
2
3       /* NN: the size of the value array
4        * SS: the size of sub-problems
5        * PP: the number of sub-problems in each step */
6       public task static local <T extends Value, NN extends int, SS extends int, PP extends
            int> IsoTask<T, T> dnc(IsoTask<T, T> sub, IsoTask<T, T> merge) {
7           final int N = NN.size;
8           final int P = PP.size;
```

```
9           final int S = N / P;
10          assert N == S * P && P >= 2 && Toolkit.isPowerOf(N, P);
11          IsoTask<T[[SS]], T[[SS]]>[PP] subs = new IsoTask<T[[SS]], T[[SS]]>[PP]();
12          for (PP i) {
13              subs[i] = Permutations.<T, T, SS>parallelize(sub);
14          }
15          IsoTask<T[[SS]], T[[SS]]>[[PP]] subtasks = new IsoTask<T[[SS]], T[[SS]]>[[PP]](subs)
                ;
16          if (N > P) {
17              return GenericPatterns.<T, N, SS, PP>dncd(subtasks, merge);
18          }
19          else {
20              return merge;
21          }
22      }
23
24      /* by default PP = 2 */
25      public task static local <T extends Value, NN extends int, HH extends int> IsoTask<T, T>
            dncD(IsoTask<T, T> sub, IsoTask<T, T> merge) {
26          return GenericPatterns.<T, NN, HH, 2>dnc(sub, merge);
27      }
```

USED IN: src/common/GenericPatterns.lime on page 1

One applicaiton would be merge sort (`http://en.wikipedia.org/wiki/Merge_sort`):

<src/examples/dnc/MergeSort.lime 26>

```
1   package examples.dnc;
2
3   import common.*;
4   import lime.util.Tasks;
5
6   public class MergeSort {
7
8       task static local <NN extends int> IsoTask<int, int> construct() {
9           final int N = NN.size;
10          IsoTask<int, int> merge = Permutations.<int, int, NN>serialize(task MergeSort.<NN>
                merge);
11          if (N > 2) {
12              final int H = N / 2;
13              return GenericPatterns.<int, NN, H>dncD(MergeSort.<H>construct(), merge);
14          } else {
15              return merge;
16          }
17      }
18
19      public static local <NN extends int> int[[NN]] merge(int[[NN]] input) {
20          NN i = 0;
21          NN H = (NN)(NN.size / 2);
22          NN j = H;
23          NN c = 0;
24          NN r = 0;
25          final result = new int[NN];
26
27          while (true) {
28              int lv = input[i];
29              int uv = input[j];
30              if (lv < uv) {
31                  result[c++] = lv;
```

```
                i++;
                if (i == H) {
                    r = j;
                    break;
                }
            }
            else {
                result[c++] = uv;
                j++;
                if (j == 0) {
                    r = i;
                    break; // NOTE: "1n" doesn't work
                }
            }
        }

//      NN r = i == 0 ? j : i;
        while (c != 0) {
            result[c++] = input[r++];
        }

        return new int[[NN]](result);
    }

    public static void main(String[] args) {
        final int N = 8;
        int[[N]] intN = Toolkit.<N>randomIntArray(System.nanoTime());
        System.out.print(Toolkit.INPUT);
        Toolkit.<int, N>output(intN);

        // computes the result
        ValueArraySink<int, N> sink = new ValueArraySink<int, N>();
        var t = Tasks.source(intN)
             => ([ MergeSort.<N>construct() ])
             => task sink.dump;
        Timer timer = new Timer();
        timer.start();
        t.finish();
        timer.stop();

        // computes the reference and compares
        int[N] reference = new int[N](intN);
        reference.sort();
        sink.compareWith(reference);
    }

}
```

Here is an example of fast Fourier transform (`http://en.wikipedia.org/wiki/Fast_Fourier_transform`):

<src/examples/dnc/FFT.lime 27>

```
package examples.dnc;

import common.*;
import lime.util.PseudoRandom;
import lime.util.Tasks;

public class FFT {

```

```
    public static final long SEED = 10101010;

    task static local <NN extends int> IsoTask<complex, complex> construct() {
        final int N = NN.size;
        IsoTask<complex, complex> merge = Permutations.<complex, complex, NN>serialize(task
            FFT.<NN>merge);
        if (N > 2) {
            final int H = N / 2;
            return GenericPatterns.<complex, NN, H>dncD(FFT.<H>construct(), merge);
        } else {
            return merge;
        }
    }

    public static local <NN extends int> complex[[NN]] merge(complex[[NN]] input) {
        final int N = NN.size;
        final int H = N / 2;
        final result = new complex[NN];
        double theta = -2.0f * (double)Math.PI / ((double) N);

        complex w = new complex(1, 0);
        complex ww = new complex(Math.cos(theta), Math.sin(theta));

        for (NN i = 0; i < H; i++) {
            result[i] = input[i] + w * input[(NN) (i + H)];
            result[(NN) (i + H)] = input[i] - w * input[(NN) (i + H)];
            w = w * ww;
        }
        return new complex[[NN]](result);
    }

    public static void sink(complex c) {
        System.out.println(c);
    }

    public static void main(String[] args) {
        final int N = 4;
        complex[[N]] complexN = Toolkit.<N>randomComplexArray(SEED);
        System.out.print(Toolkit.INPUT);

        // computes the result
        ValueArraySink<complex, N> sink = new ValueArraySink<complex, N>();
        complex[[N]] complexI = Permutations.<complex, N>bit_reverse(complexN);
        Toolkit.<complex, N>output(complexI);
        var t = Tasks.source(complexN)
             => Permutations.<complex, complex, N>serialize(task Permutations.<complex, N>
                 bit_reverse)
             => ([ FFT.<N>construct() ])

             => task sink.dump;
        Timer timer = new Timer();
        timer.start();
        t.finish();
        timer.stop();

        // computes the reference and compares
        complex[N] reference = new complex[N](complexN), temp = new complex[N];
        reference = new complex[N](Permutations.<complex, N>bit_reverse(new complex[[N]](
            reference)));
```

```
65          for (int d = 1; d < N; d *= 2) {
66              double theta = -2.0f * (double)Math.PI / ((double) d * 2);
67              complex w = new complex(1, 0);
68              complex ww = new complex(Math.cos(theta), Math.sin(theta));
69              int c = 0;
70              complex temp1, temp2;
71              for (int i = 0; i < N; i ++)  {
72                  temp1 = reference[(int<N>) i] + w * reference[(int<N>) (i + d)];
73                  temp2 = reference[(int<N>) i] - w * reference[(int<N>) (i + d)];
74                  temp[(int<N>) i] = temp1;
75                  temp[(int<N>) (i + d)] = temp2;
76                 w = w * ww;
77                  c++;
78                  if (c == d) {
79                      i += d;
80                     w = new complex(1, 0);
81                      c = 0;
82                  }
83              }
84              reference = temp;
85          }
86          sink.compareWith(reference);
87      }
88
89  }
```

References

Auerbach, J., D. F. Bacon, P. Cheng, and R. Rabbah (2010). Lime: The liquid metal programming language – language reference manual. Technical report, IBM Research Division.

Gamma, E., R. Helm, R. Johnson, and J. M. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Gosling, J., B. Joy, G. Steele, and G. Bracha (2005, May). The Java language specification, thrid edition. `http://docs.oracle.com/javase/specs/jls/se5.0/jls3.pdf`.

McCool, M., A. D. Robison, and J. Reinders (2012). *Structured parallel programming: patterns for efficient computation*. Amsterdam: Elsevier/Morgan Kaufmann.