Calculation Coverage Testing in Scientific Applications

Yoshiki Sato
Information Technology Center
The University of Tokyo
2-11-6 Yayoi, Bunkyo-ku
Tokyo, Japan
yoshiki@cc.u-tokyo.ac.jp

Shumpei Hozumi
Graduate School of Information Science and Technology
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku
Tokyo, Japan
hozumi@csg.ci.i.u-tokyo.ac.jp

Shigeru Chiba
Graduate School of Information Science and Technology
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku
Tokyo, Japan
chiba@acm.org

ABSTRACT

A typical implementation of scientific applications includes a large number of iterative calculations. For performance optimization, these calculations are often partitioned, grouped, and reordered for execution. Since this refactoring is repeatedly performed during development, it is one of the major source of bugs and thus tool support is necessary for debugging. This study discusses this problem and proposes tool support through a testing framework. This testing framework can help developers perform the tests we call *calculation coverage testing*. It investigates whether the grouped calculations cover all the calculations performed by the original (and often naively implemented) program. It also investigates whether their execution order is correct. To demonstrate this idea, we also presents *HPCUnit*, our prototype testing framework for Java, and then reports an empirical study applying it to the Java Grande Forum Benchmark Suite.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Design, Languages, Verification

Keywords

Aspect oriented programming, domain specific language, scientific computation, unit testing

1. INTRODUCTION

Typical scientific applications involve programs that analyze and solve scientific problems on a large scale high performance computer. Their computation, referred to as *stencil computation* [12], comprises iterations of massive numbers of calculations to simulate the temporal changes of a physical model. The physical model is often implemented using iterative finite-difference techniques over a spatial grid. A typical implementation of the spatial grid is a multi-dimensional array if the simulation is implemented in a programming language such as C, C++, Fortran, and Java. At each iteration for each time tick, the next value of every element of the array is calculated from its previous value and the values of its neighboring elements.

A naively implemented scientific application is a program with a simple structure, with only the mathematics in the program being complex. However, since execution performance is a crucial concern in this application domain, the programs of real scientific applications are written with various complex programming techniques for performance tuning, for example, for improving cache hit ratios and hiding communication latency. Hence, the resulting programs are often extremely complicated and debugging the implementation of these techniques consumes large amount of development time.

This study discusses tool support required for this type of debugging. First, we argue that programming techniques for performance tuning share a common idea. The kernel computation of most scientific applications applies a simple calculation to every element of a large array. Performance-tuning techniques partition the elements and group them with their neighbors. Next, they reorder the execution of the calculations applied to the elements by considering the groups. Calculations in the same group are executed together on the same node, in particular, if they exist on a distributed memory machine. In NVIDIA Compute Unified Device Architecture (CUDA), a group of 32 threads calculations composes a warp. This reordering enhances data locality and achieves a better cache hit ratio than a naive ordering, for example, iterating over the elements from first to last.

This grouping, referred to as *calculations grouping*, is a major source of difficulty in coding and debugging. Since the grouping is often complex, it can be difficult for developers to test whether the grouping is consistently implemented. A typical bug caused by calculation grouping is an incorrect specification of the boundary of a group of elements. The boundaries of some groups might overlap, or they might be separate, and some elements might not belong to any group

ISSTA'15, July 12–17, 2015, Baltimore, MD, USA

http://dx.doi.org/10.1145/2771783.2771807

(and thus the values of the elements would not be calculated at all). Such a *coverage gap* bug often occurs while the developer is modifying the program to reorder calculations. Developers must attempt various types of reordering on a trial-and-error basis to achieve the best performance.

This study proposes the need for tests to detect such a coverage gap bug, referred to as *calculation coverage testing*. Such a testing framework must support developers to ensure that the values of all the elements are calculated without duplication or loss. Such a framework has received little attention from the software engineering perspective. To illustrate an example of such a framework, we present our testing framework, named *HPCUnit*, which is our initial prototype for calculation coverage testing of Java programs. The framework allows the developer to specify in a domain-specific language, i.e., which part of the code implements the kernel calculation of each element. Next, it automatically translates the Java program to include the logging code of the kernel calculations. After the program is run, the generated logs are collected and analyzed by the Java program written by the developer to verify that the calculation grouping is correct.

The remainder of the paper is organized as follows. Section 2 shows the motivation for calculation grouping, and then describes calculation coverage gap. Section 3 introduces the idea of the calculation coverage testing and the design of the HPCUnit, which it evaluates through an empirical study on the Java Grande Forum Benchmark Suite (JGF), a suite of popular real world scientific applications. Section 4 discusses related work. Section 5 concludes the paper.

2. TECHNIQUES FOR PERFORMANCE OPTIMIZATION

First, this section presents typical optimization techniques in scientific applications and shows that they are based on the same idea we call *calculation grouping*. The typical computation of a scientific application comprises a number of small kernel calculations and optimization techniques that divide the calculations into small groups and execute every group in an appropriate order. Next, we describe calculation grouping as a frequent bug source. This type of bug is called a *coverage gap* bug since it arises from the resulting groups that do not exactly cover all the calculations.

2.1 Calculation Grouping

2.1.1 Loop Optimization

Loop optimization, also known as loop tiling, is similar to cache blocking and is a well-known calculations grouping technique for improving cache hit ratios in scientific applications. It attempts to partition the iterations of a loop for executing kernel calculations into smaller groups fitting the cache size to exploit locality. The grouped calculations are spread over both spatial and temporal dimensions until the data size required by the calculations exceeds the size of cache memory. Code 1 shows an example code of two-dimensional matrix transposition. After calculations are grouped using loop tiling, the steps of the original `i`, `j`-loops are changed from 1 to `bs`, with the result that the loops are grouped into `bs` × `bs` matrices, as shown in Figure 1. In the newly created `ii`, `jj`-loops, the read accesses to the matrix `B` are efficiently performed as long as the sequential accesses to the matrix data are within the processor cache line sizes.

```
/* Before tiling */                                          1
for (i=0; i<xsize; i++)                                      2
  for (j=0; j<ysize; j++)                                    3
    A[i][j] = B[j][i];                                       4
                                                             5
/* After tiling */                                           6
for (i=0; i<xsize; i+=bs)                                    7
  for (j=0; j<ysize; j+=bs)                                  8
    for (ii=i; ii<min(i+bs,xsize); ii++)                     9
      for (jj=j; jj<min(j+bs,ysize); jj++)                   10
        A[ii][jj] = B[jj][ii];                               11
```

Code 1: Loop tiling

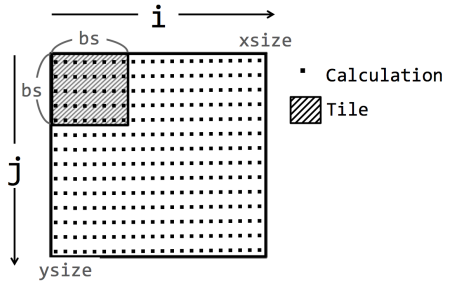

Figure 1: Calculations are grouped into a tile

A calculation grouping similar to the one seen in loop tiling is also observed in optimized code of stencil computation, which exploits cache blocking [23] or time skewing [27, 26, 20, 9]. Code 2 illustrates a simplified five-point stencil computation, which sums up the values of neighbors. After stencil computation is spatially grouped along `i`, `j`-loops similar to the previous example, the original iterations are partitioned into smaller sets of calculations by the inner-most `ii`, `jj`-loops, whose sizes are specified by `xsize` and `ysize`. Time skewing is a more effective loop tiling optimization that allows more calculations to be processed for improving temporal locality among temporal iterations. With time skewing applied, calculation groups are extended to include the calculations at different time steps. The calculation groups by `ii`, `jj`-loops in the five-point stencil code are enlarged to include the calculations executed at several consecutive time steps (the number of the steps is specified by `tsize`). Thus, some calculations will be executed before all the calculations as the previous time steps are executed.

2.1.2 Using a Third-party Library

Scientific applications implemented using third-party libraries might also involve calculation grouping. Such a scenario is found in the implementation of real-space density-functional theory (RSDFT) [44, 15] from first-principles quantum mechanical electronic structure calculations. Its core program module, called Gram-Schmidt (GS) orthonormalization, a method for orthonormalizing a set of vectors in an inner product space is implemented using the basic linear algebra subprogram (BLAS) [10] library to exploit access locality. BLAS is a set of low-level subroutines for linear algebra operations such as matrix multiplication. One of the advantages of BLAS is that it is highly tuned to achieve the best cache hit ratios using various programming techniques including loop optimization as mentioned previously. Other advantages are the maturity of its implementation and its large user base.

To use the BLAS library, calculation grouping must be modified to fit the inputs accepted by BLAS. For example, the calculations of GS orthonormalization are divided into groups processed using the matrix-multiplication function, groups using the matrix-vector multiplication function of BLAS, and

```
// Original code
for (t=0; t<timesteps; t++)
  for (i=1; i<xsize-1; i++)
    for (j=1; j<ysize-1; j++)
      C[i][j] =
        stencil(C[i+1][j] + C[i-1][j] +
                C[i][j] +
                C[i][j+1] + C[i][j-1]));

// After applying cache blocking
for (t=0; t<timesteps; t++)
  for (i=1; i<xsize-1; i+=bx)
    for (j=1; j<ysize-1; j+=by)
      for (ii=1; ii<min(xsize-1,i+bx); ii++)
        for (jj=1; jj<min(ysize-1,j+by); jj++)
          C[ii][jj] =
            stencil(C[ii+1][jj] + C[ii-1][jj] +
                    C[ii][jj] +
                    C[ii][jj+1] + C[ii][jj-1]);

// After applying time skewing
for (i=1; i<xsize-1; i+=bx)
 for (j=1; j<ysize-1; j+=by)
    for (t=0; t < tsize; t++) {
      tc = tsize - t - 1
      xmin = max(0, i-2*t);
      xmax = min(xsize+2*tc, i+bx-2*t)
      ymin = max(0, j-2*t);
      ymax = min(ysize+2*tc, i+by-2*t)
      for (ii = xmin; ii < xmax; ii++)
        for (jj = ymin; jj < ymax; jj++)
          C[ii][jj] =
            stencil(C[ii+1][jj] + C[ii-1][jj] +
                    C[ii][jj] +
                    C[ii][jj+1] + C[ii][jj-1]);
    }
```

Code 2: 5-point stencil computation

```
// Original code
for (i=0; i<N; i++)
  for (j=0; j<i; j++) {
    ip=0;
    for (k=0; k<M; k++)
      /* inner product */
      ip+=A[j][k] * B[i][k];
    for (k=0; k<M; k++)
      /* orthogonal projection subtract */
      A[i][k] -= A[j][k]*ip;
  }
  /*
   * normalization code for
   *   A[i] /= |A[i]|
   * copying vectors in i,j-loops
   */

}

// Optimized by using GEMM and GEMV
for (is=istart; is<=iend; is+=L1) {
  ie = min(is+L1-1, iend);
  il = ie-is+1;
  for (js=jstart; js<=jend; js+=L1) {
    je = min(js+L1-1, jend);
    je = min(je, je-1);
    jl = je-js+1;
    if (jl<=0) continue;
    if (is>=je+1)
      gsGemm(is,ie,js,je);
    else if (il<=L2)
      gsGemv(is,ie,js);
    else {
      gs(is,ie,js,je,max(L2/2,L1));
    }
  }
}
```

Code 3: Gram-Schmidt orthonomalization

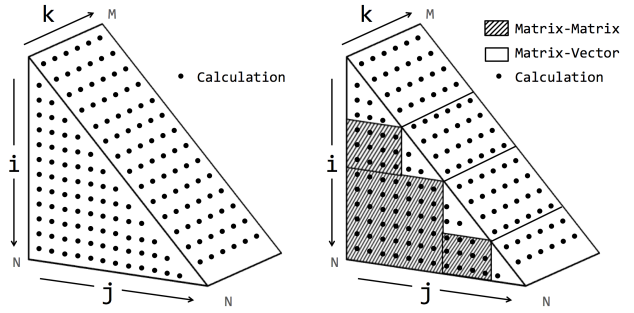

Figure 2: The calculations by the triple loops of GS orthonormalization. The right illustrates the case in which the calculations are grouped to use the BLAS library

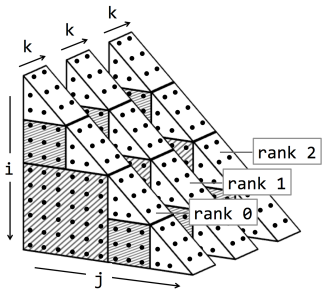

Figure 3: The calculations of GS orthonormalization are partitioned along the k-axis if the program is parallelized by MPI

others. A simple implementation of GS orthonormalization uses triple-loop calculations, as shown in Code 3, where A and B are an array of N vectors, whose size is M. Line 9 is the kernel calculation of GS. The entire calculation, an iterative calculation over a triangular prism constructed by the loop parameters i, j, and k, is illustrated in the upper figure in Figure 2. The lower figure in Figure 2 shows the calculations of the modified program shown in the lower part of Code 3. The quadrangular and small triangular prisms represent the calculations processed using the general matrix multiplication (GEMM) and the general matrix-vector multiplication (GEMV) of BLAS, respectively.

2.1.3 *Offloading*

The last calculation grouping scenario is observed in distributed computing through the Message Passing Interface (MPI),the existing standard of inter-node communication in scientific computing. In this scenario, calculations are offloaded onto multiple distributed processors. Since MPI does not provide shared memory abstraction, an array must be partitioned into sub arrays on every processing element (PE), and the calculations must also be partitioned and grouped on every PE. Figure 3 illustrates the calculation grouping of GS orthonormalization program distributed and parallelized using MPI. In the MPI, an array index for accessing array elements must be appropriately shifted on every PE, since the index of the first element of every sub array is zero. This is also a source of bugs.

Calculation grouping will be more complicated if the stencil computation with loop tiling is offloaded over multiple PEs using MPI. Since the calculations on the boundary of each tile must access values of elements of adjacent tiles, which might be located on remote PEs, a naive implementation will cause the calculations to stall as a result of communication latency for fetching values from remote PEs. A well-known technique for hiding this communication latency is to use a shadow region. The sub array on each PE is enlarged to hold copies of elements on adjacent PEs required to calculate the PE border elements. The region holding the copies is called a shadow region as shown in Figure 4. Next, at every time tick, the values in the shadow region are asynchronously fetched from the adjacent regions and, while waiting for the completion of the fetching, the calculations that do not require the fetched values are executed first. Calculations depending on the fetched values are executed last. Overlapping some calculations with data communication hides the communication latency, but the calculation grouping must be modified. Code 4 shows the code for stencil computation so that calculation will be overlapped with communication.

Offloading some of the calculations onto hardware accelerators, such as a General Purpose Graphic Processing Unit

```
for (i=1; i<xsize-1+2*sw; i++)
  for (j=1; j<ysize-1+2*sw; j++)
    if ((i >=1 && i < sw) ||
        i >= xsize+sw && i <xsize-1+2*sw)
      stencil_shadow(1,sw,xsize+sw,xsize-1+2*sw)
    else
      C[i][j] =
        stencil(C[i+1][j] + C[i-1][j] +
                C[i][j]
                C[i][j+1] + C[i][j-1]);

void stencil_shadow(int xs, int xe,
                    int ys, int ye) {
  /* Exchanges array data with
     neighboring PEs in parallel */
  exchange_data(C, xs, xe, ys, ye);

  for (i=1; i<xsize-1+2*sw; i++)
    for (j=1; j<ysize-1+2*sw; j++)
      if ((i >=1 && i < sw) ||
          i >= xsize+sw && i <xsize-1+2*sw)
        C[i][j] =
          stencil(C[i+1][j] + C[i-1][j] +
                  C[i][j]
                  C[i][j+1] + C[i][j-1]);
}
```

Code 4: The stencil computation where calculation is overlapped with communication

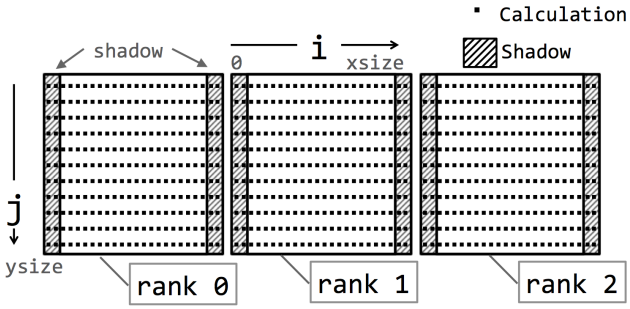

Figure 4: Every PE allocates a shadow region and holds copies of border elements of adjacent PEs

(GPGPU) also requires modifying the calculation grouping. For example, NVIDIA CUDA [29] has multiple streaming multi-processors (SMs) containing multiple CUDA cores for SIMD operations. Every SM has local memory space shared among its CUDA cores. To exploit this architecture, calculations must be partitioned as in the case of the MPI. Furthermore, memory coalescing is important in obtaining the best performance in GPGPU programming, but it requires a specialized calculation grouping, i.e., loop tiling. The calculations distributed to every SM are divided into several warps of 32 threads each. A warp can access every successive 128 bytes of memory in a single transaction. Hence, calculations accessing successive memory must be grouped and allocated to the same warp. Code 5 shows the implementation of matrix multiplication in CUDA. `matmul` is a function executed by each thread, which calculates a single element of the resulting matrix `C`. The second `matmul` function considers shared memory and memory coalescing. First, it loads several successive elements of `A` and `B` onto shared memory `_A` and `_B` in collaboration with other adjacent threads. Next, it calculates an element of `C` and writes it considering memory coalescing.

2.2 Coverage Gap Bugs

As shown above, implementing complex calculation grouping is error-prone. Next, we mention possible bugs relevant to calculation grouping. We call them coverage gap bugs. Since optimization techniques are gradually applied to a program,

```
// Matrix multiplication
__global__ void
matmul(float* C, float* B, float *A, int N) {
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  int x = blockIdx.x * blockDim.x + threadIdx.x

  float tmp = 0;
  for (k=0; k<N; k++) tmp += A[y*N+k] * B[k*N+x];
  c[y*N+x] = tmp
}

// Matrix multiplication with memory coalescing
__global__ void
matmul(float* C, float* B, float *A, int N) {
  int by = blockIdx.y * ytile;
  int bx = blockIdx.x * xtile;
  int y = threadIdx.y;
  int x = threadIdx.x;

  float tmp = 0;
  __shared__ float _A[ytile][xtile];
  __shared__ float _B[ytile][xtile];

  for (bk=0; bk<N; bk+=xtile) {
    _A[y][x] = A[(by+y)*N+bk+x];
    _B[y][x] = B[(bk+k)*N+bx+x];
    __syncthreads();
     for (k=0; k<xtile; k++) tmp += _A[y][k]*_B[k][x];
  }
  __syncthreads();
  C[(by+y)*N+bx+x] = tmp;
}
```

Code 5: The CUDA kernel of matrix multiplication

often on a trial-and-error basis, a developer must implement multiple types of grouping during the development. This increases the risk of creating bugs.

2.2.1 Calculation Leaks

A calculation leak leads to a severe but common bugs in iterative calculations. This is likely to occur because of errors in an initial value, a termination condition, or a step value of a loop. If a calculation is included in an incorrect group, it might not be critical, as it will be processed sooner or later. Specifying an incorrect calculation group size in CUDA kernel code will cause a calculation leak, but this will merely result in lower memory bandwidth utilization. Furthermore, in the case of a communication-overlapping example, it may only cause more MPI communication required than the developer had expected.

However, if no other group covers leaked calculation, the application will never obtain the result of that calculation. A typical kernel calculation in scientific applications reads neighboring elements of an array as a five-point stencil calculation needs the results of top, bottom, right, left, and itself. This indicates an incorrect calculation result propagating throughout the calculations. Therefore, even a small calculation leak could lead to a totally incorrect application program result. Let us return to the matrix transposition example in Code 1. If the matrix size defined by `xsize` and `ysize` has a gap with tile size `bs` in other words, if it is not a multiple of the tile size, and the loop tiling is implemented without the two inner loops at lines 9 and 10 in Code 1, then the program will be as follows.

```
for (i=0; i<xsize; i+=bs)
  for (j=0; j<xsize; j+=bs)
    A[i][j] = B[i][j];
```

Some calculations will be leaked and never executed. This is a simple example, but this type of calculation leak is easy to reproduce.

2.2.2 Duplicated Calculations

An overlap between calculation groups causes a duplicated calculation. It is an opposite type of coverage gap. In general, similar human errors on loop optimization create this gap through the development process. On the other hand such a

coverage gap is not always harmful. It is sometimes purposely produced in a case such as communication overlapping. As mentioned in the previous section, a shadow region contains duplicated calculations containing copies of calculations from the neighboring PEs. The primary purpose of this duplication is to reduce frequent communication for data exchanges between PEs. Next, which calculation belongs to which group will be designed with the utmost care.

On the other hand, an incorrect calculation duplication that developers do not intend wastes computational resources, such as processors, memory, and network bandwidth. For instance, returning to the matrix transposition example again, if the calculation group of the two innermost loops created for the loop tiling in Code 1 has incorrect termination conditions, as follows.

```
for (j=0; j<ysize; j+=bs)                                    7
  for (i=0; i<xsize; i+=bs)                                  8
    for (ii=i; ii<xsize; ii++)                               9
      for (jj=j; jj<ysize; jj++)                             10
        A[ii][jj] = B[jj][ii];                               11
```

then the tiled calculations always reach the boundary of the matrix. As a result, the number of executions of the kernel calculation is estimated to be `xsize` × `ysize` × *the number of tiles*, where the number of calculations amounts to the greater of either `xsize/bs` or `ysize/bs`. This type of duplication cannot be regarded as a bug, because it does not result in an incorrect value in each calculation. However, the original motivation of grouping calculations is to improve runtime performance. Such unnecessary execution of duplicated calculations without any benefit is wasteful code modification.

2.2.3 Out-of-Order Calculation

The last type of bug discussed herein is the out-of-order calculation, occasionally caused by coverage gaps, such as calculation leaks and duplications. An advanced optimization for improving cache hit ratios is performed by reordering calculation groups and their member calculations. For instance, when program execution reaches the end of the tile, loop tiling changes the order of each calculation to align them in the orthogonal direction. Time skewing causes more aggressive changes in calculation order, which exchanges a calculation beyond an outer-most loop controlling each time step.

Out-of-order calculations are frequently observed in scientific applications exploiting optimization by calculation groups. The use of the BLAS library often leads to out-of-order calculations, for example, when matrix functions GEMM and GEMV are used together, the calculation groups are modified to fit the inputs to the functions.

Incorrect calculation orders can cause a performance drawback. For example, access locality to arrays may decrease compared to the expectation of the developer. Let us consider the stencil calculation loop tiling example again in the middle of Code 2. If the order of the iterations of the `ii`, `jj`-loops is changed as follows.

```
for (jj=1; jj< min(ysize-1,j+by); jj++)                      13
  for (ii=1; ii<min(xsize-1,i+bx); ii++)                     14
    C[ii][jj] =                                              15
      stencil(C[ii+1][jj] + C[ii-1][jj] +                    16
              C[ii][jj] +                                    17
              C[ii][jj+1] + C[ii][jj-1]);                    18
```

then the iterations will not perform well since the successive accesses in the `jj`-loop are not performed on contiguous array elements if the two dimensional array is alined in the manner of C, rather than Fortran. This will decrease the cache hit ratio.

A worse scenario of out-of-order calculations is observed in the time skewing example at the bottom of Code 2. If the time order is reversed.

```
for (t = tsize-1; t >= 0; t--) {                             23
  tc = tsize - t - 1                                         24
  xmin = max(0, i-2*t);                                      25
  xmax = min(xsize+2*tc, i+bx-2*t)                           26
  ymin = max(0, j-2*t);                                      27
  ymax = min(ysize+2*tc, i+by-2*t)                           28
  for (ii = xmin; ii < xmax; ii++)                           29
    for (jj = ymin; jj < ymax; jj++)                         30
      C[ii][jj] =                                            31
        stencil(C[ii+1][jj] + C[ii-1][jj] +                  32
                C[ii][jj] +                                  33
                C[ii][jj+1] + C[ii][jj-1]);                  34
}                                                            35
```

then the order of the calculations in the `ii`, `jj`-loops is changed, and some calculations are executed with incorrect values of neighboring array elements, which have not been updated. This will result in an incorrect output by the application program.

3. CALCULATION COVERAGE TESTING

Our observation is that the problems mentioned in the previous section result from coverage gaps, which are incorrectly produced when developers create groups of iterative calculations and then reorder them for runtime optimization. Optimized implementation by grouping and reordering calculations is difficult verify without knowing the developers' intention, since coverage gaps are sometimes prepared with a clear purpose, such as duplication for a shadow region to hide the latency of communication, as previously described.

Until now, this type of verification has been conducted by developers who look over the entire result of calculations manually or automatically. Manual checking of an enormous number of results is beyond human recognition capability, but several researchers have actually attempted to detect incorrect calculation coverage by detecting error results while reducing problem size. Writing script code that screens aberrant results, rather than screening it by eye, is a reasonable approach, but, it is insufficient because the results of the original and the optimized kernel calculations may differ if the calculations include floating-point arithmetic and/or data initialization with random numbers as pointed out in [5].

Our proposal is to provide a coverage testing tool specialized for validating calculation coverage, not focusing on an implementation of each calculation. This approach is similar to but different from code coverage testing, which measures what percentage of subroutines or code statements have been exercised by a test suite. It does not consider functional aspects of a program, but instead, measures code coverage while launching several separately developed test operations. Our proposed calculation coverage testing process also checks whether a calculation group correctly covers parameter spaces. We assume that coverage of iterative calculations is always preserved as long as it is optimized by regrouping and reordering. We assume that a set of calculations never changes even if complicated loop tiling, which produces entirely different calculation groups, is applied. Under this assumption, coverage gaps could be detected by equivalence checking of calculation groups between an optimized program and an original one.

Our idea is that a calculation can be identified with its runtime context *log* during the execution, which includes the actual values of parameters, such as function parameters and, local and global variables. Therefore, the calculation coverage testing tool must reify an abstract model of calculation groups in set form. To make it easy to compare the reified model

with the model corresponding to the correct grouping, the testing tool must provide basic set operations such as union, intersection, difference, and subset. It must also provide advanced set operations including filter, map, and fold. Next, the testers can define an invariant that is to be preserved to avoide leaks, duplication, and incorrect calculation orders.

To further investigate this idea, we have developed a testing framework, *HPCUnit*, a prototype framework for Java programs. We present it in the rest of this section.

3.1 HPCUnit

HPCUnit is a unit-testing framework for supporting a calculation coverage test in Java. It is implemented as an extension to JUnit [13], a popular unit-testing framework, and thereby allows developers to check calculation coverage, along with ordinary function testing by JUnit. The HPCUnit provides for developers' annotations and helper classes to describe which method is a kernel function and which invocations to the kernel functions are logged with their arguments to construct a calculation group. A test method for the HPCUnit takes several sets as parameters. Each parameter is an ordered set of log entries representing a calculation group. Its definition is given by an annotation attached to the parameter. A typical test method takes two sets as parameters, one representing the calculation group obtained during the execution of an original intuitive program and the other representing those obtained during the execution of a program optimized by reordering calculations. The test method compares these sets given as arguments and verifies that the calculation groups do not involve a coverage gap. The HPCUnit provides helper classes for comparison and verification.

Here, we present an example of test code for the HPCUnit in Code 6. It tests the optimized GS orthonormalization program in Figure 2. The test code contains a test-driver method (lines 6-10) and a test method (lines 14-28). The test driver is annotated to launch the tested program before running coverage testing (lines 6-8). The parameters to the test method are also annotated to specify the calculation group represented by the parameter (lines 15-17, 18-20, and 21-23). The annotations for the parameters can describe a complex rule for logging the invocations of the kernel functions only in the specific contexts, such as within a particular control flow and within a particular method body. When the tested program is launched, the logging code is embedded in the program by the HPCUnit before the execution, so that the specified invocations of the kernel functions are recorded as logged calculations. Then, the test methods are individually executed to investigate the logged calculations passed as arguments (lines 26, 27).

3.1.1 Annotations for Collecting Logs

The calculation groups are declaratively specified with annotations. The specifications are given as arguments to the annotations and they are described in a syntax similar to set-builder notation in mathematics. The properties that the set elements satisfy are described in a pointcut-like language in AspectJ [21], as shown in Table 1. The properties specify which method invocations are logged with which arguments and target. Each invocation is recorded as a tuple of the specified values and the tuple is an element of the set representing a calculation group.

```
/* Test suite declaration */                                          1
@RunWith(HUTestRunner.class)                                          2
class GSTest {                                                        3
                                                                      4
/* Test driver */                                                     5
@HUBeforeClass void testRun() {                                       6
  @HUTarget GS gs1 = new NormalGS();                                  7
  @HUTarget GS gs2 = new OptimizedGS();                               8
  gs1.calc();                                                         9
  gs2.calc();                                                         10
}                                                                     11
                                                                      12
/* Test method */                                                     13
@HUTest public void testCoverageGS (                                  14
  @HUTrace("{ (i,j,k) |                                               15
    call(void kernel(int i, int j, int k))                            16
      && within(NormalGS)}") HUList correct,                          17
  @HUTrace("{ (i,j,k) |                                               18
    call(void kernel(int i, int j, int k))                            19
      && cflow(call(dgemm(..))}") HUList squares,                     20
  @HUTrace("{ (i,j,k) |                                               21
    call(void kernel(int i, int j, int k))                            22
      && cflow(call(dgemv(..))}") HUList triangles)                   23
{                                                                     24
  HUList nullset = HUList.getNull();                                  25
  assert(correct,is(squares.union(triangles)));                       26
  assert(nullest,is(squares.intxn(triangles)));                       27
} }                                                                   28
```

Code 6: A test example with HPCUnit

Table 2: Reserved words available in intensional definitions

Word	Description
count	Executed number
time	Started time
threadID	ID of thread
mpiRank	Rank number of MPI
mpiSize	Size of MPI processes

In Code 6, the definitions of calculation groups are described as the arguments to the `HUTrace` annotations. The method parameters `correct`, `squares`, and `triangles` of the test method `testCoverageGS` are annotated with `HUTrace`. While the tested method `calc` in `NormalGS` or `OptimizedGS` class is running, the arguments `i`, `j`, and `k` to the `kernel` method called within the `NormalGS` class, within the control flow of the `dgemv` method, or within the control flow of the `dgemm` method, construct a tuple and they are stored in the `HUList` objects `correct`, `squares`, or `triangles`, respectively. The parameter `correct` represents the calculation group illustrated by the left prism in Figure 2, whereas `squares` and `triangles` represent the group denoted by the matrix-matrix and matrix-vector in the right prism, respectively.

A member of the tuples stored in the `HUTrace` objects can be a method argument or a target object. In the current design of HPCUnit, the values of local variables or object fields cannot be logged. Thus, all the values needed for constructing a tuple have to be explicitly passed to the kernel function as arguments.

Besides the method arguments, a few reserved keywords are available for obtaining other types of useful values in the running execution contexts as listed in Table 2. For instance, if an application program is an MPI program, the processor-element number is bound to the reserved keyword `mpiRank`. The following annotation specifies a set of tuples with four elements.

```
@HUTrace("{
  (mpiRank, i, j, k)
      | call(void kernel(int i, int j, int k)) }")
```

The first element is the processor-element number and the other elements are the arguments passed to the `kernel` function.

3.1.2 Coverage Testing APIs

The test method receives as arguments and investigates the sets of tuples representing calculation groups by calling methods in the `HUList` class. These methods implement basic set operations such as union and intersection as listed

Table 1: Pointcut-like annotations in HPCUnit

Annotation	Description
call (*MethodPattern*)	Kernel calculations of method invocations specified by *MethodPattern*
within (*TypePattern* | *MethodPattern*)	Kernel calculations in a type matched by *TypePattern* or in a method of *MethodPattern*
cflow (*Annotation*)	Kernel calculations in a control flow of any calculation specified by *Annotation*
receiver (*TypePattern*)	Kernel calculations where the currently executing object is an instance of *TypePattern*

in Table 3. Some methods are used to create an empty set or a set representing a simple calculation group. In Code 6, the test method verifies that the union of `squares` and `triangles` matches `correct` and `squares` and `triangles` do not have any intersection (lines 25-27). Successful verification reveals that the calculations have been consistently reordered between the original program and the optimized program.

The `HUList` class provides the `map` (flat map) method, which applies a given operation to each element of the set and returns a flattened set. It can be used for creating a subset of the given set. The following code uses `map` to compute a subset whose elements are tuples recorded during the first iteration (*i.e.*, if the first element `count` is `1`), the code is as follows:

```
@HUTrace("{
  (count, i, j, k)
       | call(void kernel(int i, int j, int k)) }")

gs_first = gs_all.map(
  new HUMapper<HUTuple4<..>,HUTuple3<..>>(){
    HUTuple3<..> apply(HUTuple4<..> t) {
      if (t0 == 1) return new HUTuple3<>
        (t.el1, t.el2, t.el3);
    }});
```

Note that the resulting set holds tuples with three elements, where `count` is a reserved keyword referring to the iteration count of the program execution.

The `map` method is also useful when testing an MPI program. As described in section 2.1.3, a scientific application to be distributed using the MPI has separate calculation groups for each PE. Thus, the calculation group of the original non-distributed program must be compared with the union of all the groups for each PE. However, the array indices in the tuples must be appropriately shifted before computing the union. The `map` method is useful for shifting the indices. Assume that the GS program processes vectors with `k` elements each and that the length of the vectors `k` is nine million, as shown in Figure 2. Next, assume that the program is modified to use MPI, as in Figure 3, and that every vector is divided into three sub-vectors allocated on a different PE. In a typical MPI program, the range of `k` on every node is from *0* to *300,000*. Hence, when computing the union of the calculation groups, the loop index `k` recorded in a tuple has to be shifted by the offset *rank* × *300,000*. This can be implemented as follows:

```
gs = gs_mpi.map(
  new HUMapper<HUTuple4<..>,HUTuple3<..>>(){
    HUTuple3<..> apply(HUTuple4<..> t) {
      return new HUTuple3<>
        (t.el1, t.el2, t.el3 + t.el0 * 300000 );
    }});
```

The set `gs_mpi` contains four-element tuples, with the first element being `mpiRank`. Using the `map` method, we show the complete test code in Code 7 for testing an MPI program previously in Figure 4. The code verifies that the program involves neither calculation leaks nor duplications.

The `HUList` class also provides the `fold` method, which performs a reduction operation. A `HUList` object represents an ordered set, where the tuples are ordered using their recorded time. The `fold` method is used to investigate whether the order of tuples satisfies given properties. This investigation is useful for detecting a performance bug due to losing memory access locality. For example, for better

```
for (int n = 0; n < mpiSize; n++) {                              1
  rank = n;                                                      2
  HUList r = mpiResults.get(n).map(                              3
  new HUMapper<HUTuple3<..>,HUTuple2<..>>(){                     4
    HUTuple2<..> apply(HUTuple3<..> t) {                         5
      if (t.el0 == rank)                                         6
        return new HUTuple2<>(t.el1+t.el0*xsize,t.el2);          7
    }                                                            8
  });                                                            9
  if (rank > 0) {                                                10
    ovlp = ovlp == null ? r                                      11
    : ovlp.union(nonovlp.intxn(r));                              12
  }                                                              13
  nonovlp =                                                      14
    nonovlp == null ? r                                          15
    : nonovlp.diff(ovlp).union(r.diff(ovlp));                    16
}                                                                17
                                                                 18
assert(nonovlp.union(ovlp), is(correct));                        19
assert(nonovlp.union(ovlp)).diff(correct), is(empty));           20
```

Code 7: Coverage testing for shadow regions

memory-access locality, GS orthonormalization requires that the calculations be executed in descending order on `i`, as in Figure 2. The test method executes the following code.

```
gs.fold(
  new HUFolder<HUTuple3<..>,HUTuple3<..>>() {
    boolean apply(HUTuple3<..> org,
                  HUTuple3<..> cur) {
      return org.el1 > cur.el1;
    }});
```

The anonymous class of `HUFolder` implements a function for testing that every pair of adjacent tuples in the set preserves some property. The `apply` function above examines that the value of `i` is descending. The `fold` method returns true if all the tests succeed. This example can also be extended to compute the interval between the indices of adjacent calculations and to evaluate the memory-access locality. If the average interval is shorter, the locality is better and improves the cache hit ratio.

3.1.3 Integration with Other Development Tools

The HPCUnit has been developed to be integrated with JUnit. An advantage of integration with JUnit is that the test code of calculation coverage can coexist with other ordinary unit tests. The calculation coverage testing must be performed after the tested application program passes most unit tests. JUnit provides a mechanism for incorporating a user-defined test runner and the `RunWith` annotation is shown in Code 6 (line 2). The HPCUnit intercepts the loading process of the classes of the tested program and then loads them using a dedicated class loader so that the logging code will be embedded in them. Since this dedicated class loader is used only by the HPCUnit, other normal unit tests run with the tested program without logging code embedded in them. No runtime penalties because of logging are involved when the normal unit tests are running.

The HPCUnit also provides several MPI-specific annotations. Developers can select whether calculation coverage testing is performed on every PE as distributed computing or on the central PE, such as the PE with rank zero. It is selected by `HUDistributedTest` or `HUGatheredTest` instead of the `HUTest` annotation. The test on the central PE is easy to write since the HPCUnit automatically gathers all the sets on the central PE from the other PEs if the centralized test is selected. On the other hand, the distributed test can deal with larger size sets, which will be generated when the tested

Table 3: Set operations in the HUList APIs

Log Manipulation API	Description
boolean equals(Object o)	Checks equivalence against a given log object.
HUList⟨T⟩ union(HUList⟨T⟩ logs)	Returns an union set against a given log object
HUList⟨T⟩ intxn(HUList⟨T⟩ logs)	Returns an intersection set against a given log object
HUList⟨T⟩ diff(HUList⟨T⟩ logs)	Returns a difference set against a given log object
⟨U⟩ HUList⟨U⟩ map(HUMapper⟨T, U⟩ mapper)	Applies a given method defined in `HUMapper`
⟨U⟩ U fold(HUFolder⟨T, U⟩ folder, U origin)	Recombines logs using a given method defined in `HUFolder`
Log Creation API	
static ⟨U⟩ HUList⟨U⟩ getNull ()	Creates a null set
static HUList⟨HUTuple1⟨Integer⟩⟩ getLine(int L1)	Creates a zero to L1 line segment
static HUList⟨HUTuple2⟨Integer, Integer⟩⟩ getSquare (int L1)	Creates a zero to L1 square
static HUList⟨HUTuple2⟨Integer, Integer⟩⟩ getTriangle (int L1, int L2)	Creates an L1×L2 rectangular triangle
static HUList⟨HUTuple3⟨Integer, Integer, Integer⟩⟩ getCube (int L1)	Creates an L1 cube
static HUList⟨HUTuple3⟨Integer, Integer, Integer⟩⟩ getTriangularPrism (int L1, int L2, int L3)	Creates an triangular prism which face is L1×L2 rectangular triangle, length is L3

application program performs a large number of calculations. Furthermore, the HPCUnit supports mock testing. If the `HUSkip` annotation is added with the `HUTrace` annotation, the `HUList` object passed to the test method is constructed according to the annotation without executing the tested program. This is useful when the developer wants to unit-test an MPI program without running it in a distributed environment.

3.2 Empirical Study with Benchmarks

This section reports an empirical study to evaluate how calculation coverage testing is applied to a real world scientific application. We use JGF [35] developed by the Edinburgh Parallel Computing Centre (EPCC) group at the University of Edinburgh, which includes several applications aimed at evaluating the performance of typical implementations of scientific applications. It provides not only a sequential program, but also a parallel one with shared memory (multi-threaded), and one with distributed memory, where the kernel calculations are partitioned depending on the number of processing threads or MPI processes, since it has been developed to measure different execution environments. The distributed version is based on MPJ, an MPI-like message passing interface for Java mentioned in Section 2.1.2. This empirical study has been practiced in the following steps.

(a) Extracting a kernel calculation as a method named `HUKernel` from each benchmark application because of the limitation of the current HPCUnit implementation, which only recognizes a method known as kernel calculations logged for constructing a calculation group. If a kernel calculation is composed of a set of sub-kernel calculations, each block of code is extracted as a different method with an extra parameter used as a tag for distinguishing sub-kernels.

(b) Defining calculation groups of all benchmark applications. Every group can be simply specified with a tuple of method arguments extracted during execution. Note that the logs for sub-kernels include the values of an additional tag, such as `{(tag,i,j)|call(void HUKernel(int tag,int i,int j)}`) such that it is easily filtered at the test step.

(c) Developing two types of unit test to prove that parallel and distributed versions of the benchmark have no calculation coverage gap compared to the sequential one.

(d) Measuring the runtime overhead caused by collecting logs, and the execution time of the test spent for comparing logs and verifing calculation groups between different versions.

Table 4 lists the number of lines of code (LoC) of eight benchmark applications and their test code. The LoC of the test code is approximately 30 with six annotations. The test code includes a test driver to launch benchmarks in sequential, parallel, and distributed manners. The test methods for comparing logged data are nearly identical, with only the annotations being unique to consider the number and the type of the tuples representing kernel calculations. The test code of the distributed version of the programs includes boilerplate code for initialization and finalization of MPI processes.

Through the empirical study with JGF, we found duplication of calculations in both the parallel and distributed versions of MD, whereas all the other applications passed the test. MD is a molecular dynamics simulation that models particles interacting with each other under a Lennard-Jones potential in a cubic spatial volume. A kernel calculation of MD is composed of multiple sub-kernels, such as moving particles, computing forces, updating forces, and velocities. We confirmed that one of the sub-kernels for calculating the average of velocities is redundantly processed by all the threads on all the particles. To avoid the redundancy, each thread must calculate the value by accessing only its local particles. To double-check a coverage gap discovered in the test, we modified the step value of this sub-kernel loop to eliminate the gap. The modified MD simulation produced the same result as the original one. This fact shows that the coverage gap we found causes redundant runtime calculations. This would explain why this coverage gap has long been overlooked..

We measured the runtime overheads and the execution time of the calculation coverage tests for JGF. The logging code written with the HPCUnit API was embedded in the benchmark programs by the AspectJ compiler instead of the HPCUnit translator. This is done to avoid a bug in the current implementation of the HPCUnit system. All the experiments were measured on OpenJDK 1.7.0_25 64-bit with server JIT, running on a machine with Linux kernel 2.6.32. The machine had dual Intel Xeon E5-2687W processors eight cores

Table 5: Runtime overheads incurred by collecting logs and test execution time

Benchmark	Calculation size	Sequential with log[s]	Overheads [%]	Parallel with log[s]	Overheads [%]	Distributed with log[s]	Overheads [%]	Test for Parallel[s]	Test for Distributed[s]
Series	999,999	704.72	0.05	59.34	1.96	59.97	0.99	0.18	5.16
LUfact	1,652,472	4.06	55.58	4.75	1689.26	3.04	76.09	0.78	11.14
SOR	399,200,400	160.04	8460.14	748.59	509579.66	32.35	2121.52	41.22	143.35
Crypt	12,500,000	3.21	69.88	9.93	5458.99	2.35	513.25	1.3	26.82
Sparse	500,000,000	49.52	1368.23	230.21	77478.43	20.77	49.37	17.11	48.5
MD	409,600	13.32	0.92	14.11	285.92	3.59	61.27	0.13	1.27
MC	60,000	8.77	0.01	1.31	31.71	8.07	8.57	0.06	0.41
RT	250,000	12.61	3.13	1.47	20.54	2.67	47.01	0.11	1.87

Table 4: The number of lines of code of benchmark applications and test code

Benchmark	Sequential	Parallel	Distributed
Series[1]	356	440	449
LUfact[2]	721	1,092	895
SOR[3]	206	344	415
Crypt[4]	613	708	720
Sparse[5]	221	337	317
MD[6]	625	915	697
MC[7]	3,115	3,181	3,203
RT[8]	1,265	1,483	1,346
Annotation + Test code		6+30	6+35

[1] Fourier coefficient analysis
[2] LU Factorization
[3] Successive over-relaxation
[4] IDEA encryption
[5] Sparse Matrix multiplication
[6] Molecular Dynamics simulation
[7] Montecarlo Simulation
[8] 3D RayTracer

each. The command line parameters for launching Java were `-Xmx28g -Xms28g -Xmn24g -XX:SurvivorRatio=10`. We ran all the benchmarks with the maximum data size given by the suite. The parallel versions of the benchmarks ran with 16 threads, while the distributed versions ran with 16 MPI processes on a single node. Table 5 lists the results. All the data are the average values of ten repetitions, ignoring both maximum and minimum results. The results show that the performance penalties of SOR and Sparse are extremely large, since these benchmarks execute a large number of small kernel calculations. The penalties mostly depend on the number of calculations to be collected for logging. Furthermore, it was observed that the runtime overheads of the parallel versions of Series, LUFact, SOR, Crypt, Sparse, and MD were greater than their sequential versions. The primary reason for the extra overheads is that the log data collected during the execution are stored in the shared object in the current HPCUnit implementation. This fact causes non-negligible synchronization costs for accessing the object. The distributed versions of the benchmarks showed relatively smaller overheads than the parallel ones since the logs were stored in separate memory spaces although they involved overheads because of network communication. Regarding the test execution, the elapsed time of the calculation coverage testing, excluding the logging time, would be acceptable in practice for both parallel and distributed versions. The log object in the current HPCUnit implementation is a multiple set object of Apache Commons Collections^{TM} called Bag. Instead of a large Java object, using a primitive array would reduce the runtime overheads and the cost of test execution. However, that requires changing the HPCUNit API design, which we carefully designed for application development productivity. We plan to apply several optimizing techniques to reduce overheads for collecting runtime logs [37] and to reduce the Java collections memory footprint [13].

From the overall study using realistic application benchmarks, we found that writing only a relatively small amount of code for parallelization significantly improves the execution performance. The coverage testing code by the HPCUnit is even smaller than the parallelization code. For example, the parallel version of the Series benchmark is 12 times faster on a 16-core machine than on the sequential one, but the parallel version includes only 120 (440 - 356 + 36) extra lines of code, including the test code.

4. RELATED WORK

There have been a number of approaches to verify specification and runtime behaviors of application software in several research areas. Those general verification mechanisms possibly have the potential to be used for checking coverage gaps by verifying the behavior of an application itself. Our main contribution is to raise the necessity of a domain-specific testing approach and tool support, which can reduce the development costs for prototyping, tuning, and testing of scientific applications that mostly comprises iterative calculations.

According to recent surveys of testing tools in scientific applications [31, 43], various testing frameworks are currently available even for the high-performance computing (HPC) community [39, 38, 2, 11]. A unit-testing framework such as JUnit [7] is used to test individual program modules of the source code to ensure that they work as the developers intend. In particular, the National Aeronautics and Space Administration has been developing two unit testing frameworks pFUnit and FUnit for Fortran 95 based scientific applications [39, 38]. The aims of both tools are to provide a port of JUnit supporting the ability to launch MPI tests and report results back as a single test. It might be possible to perform calculation coverage testing on top of these tools. However, since it has no dedicated supports for calculation coverage testing, this will require developers to write a substantial amount of code to collect, record, and compare runtime context logs for each calculation. The same discussion can be applied to code coverage test tools for HPC [14, 24].

Model checking [8] is widely recognized as the primary technique for verifying the behavior of concurrent programs. In fact, several model-checking approaches have been applied in HPC, although their main focus is to verify the correctness of the MPI-based parallel/distributed program design [31, 25, 33, 32, 34, 30, 40, 41]. Previous studies using the SPIN model checker [17] found it difficulty to create an accurate model of the program being verified [25, 33, 32, 34]. In recent studies, several model-checking approaches directly verify the MPI programs [30, 40, 41]. These studies are useful for verifying that all the potential interleavings of the MPI programs work without errors.

Runtime verification aims at combining testing with formal methods, verifing at runtime the behavior and histories of program events. A large number of runtime verification approaches such as MaC [22], PathExplorer [16], MOP [3], and Tracematches [1] have been developed. These approaches enable validation using code automatically generated from formal specifications and then invoking user-defined code at any point in a program at runtime. In principle, the specification, which is usually described using a domain-specific language, can refer to historical information at a specified program point. Therefore, some runtime verification tools could be used for calculation coverage testing. However, they have not been clearly reported to have sufficient capability to validate the histories of specified arguments within a specified program control-flow through comparison with others.

Metamorphic testing [4], first proposed by Chen, attempts to verify *non-testable programs* [42] including numerical and scientific computations, for which testers can not decide whether the results of a program are correct. Instead of verifying a test output of the target function, metamorphic testing generates follow-up test cases based on domain-specific properties (in particular, metamorphic relations in the context of metamorphic testing), and then verifies whether a set of their outputs is correct, which has been applied to various application domains, such as machine learning [28], bioinformatics [6], and web services [36]. Chen et. al. have also applied metamorphic testing to partial differential equations since it is difficult to validate the correctness of outputs from that type of numerical program because of truncation and rounding errors [5]. They identified an invariant as a metamorphic relation preserved even if the densities of mesh grids increase. Calculation coverage could be regarded as a new metamorphic relation in scientific applications, which requires a special treatment to record and manipulate the histories of specified context data of a program.

The preliminary work of this research has been presented at local workshops where the papers are not reviewed or published [18, 19]. This study is an extended version of these papers to discuss the fundamental idea behind the HPCUnit and a case study.

5. CONCLUSION

This study introduced calculation grouping for optimizing scientific applications and problems resulting from coverage gaps caused by partitioning and reordering to improve the runtime performance. We pointed out the necessity of calculation coverage testing in addition to ordinary software testing, such as unit and code coverage testing. To demonstrate this idea, we presented a Java-based testing framework, the HPCUnit. We illustrated concrete examples of test code for detecting several calculation gaps. Our preliminary experiment showed that the runtime overheads and memory consumption were one of the primary topics to be considered for developing a practical framework for calculation coverage testing. Although calculation coverage testing focuses only on iterative calculations, it is an essential aspect of scientific applications from the programming perspective.

6. ACKNOWLEDGMENT

We would like to express our deep gratitude to the anonymous reviewers. Jun-Ichi Iwata provides us his RSDFT code and gives a lecuture on that. This research was partly supported by the CREST program of JST and JSPS KAKENHI Grant Number 26870165.

7. REFERENCES

[1] C. Allan and P. Avgustinov. Adding trace matching with free variables to AspectJ. *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2005.

[2] J. S. Anil Kumar. CUnit. http://cunit.sourceforge.net, 2004.

[3] F. Chen and G. Roşu. Mop: An efficient and generic runtime verification framework. *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2007.

[4] T. Chen, S. Cheung, and S. Yiu. Metamorphic Testing: A New Approach for Generating Next Test Cases. *Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology*, 1998.

[5] T. Chen, J. F. J. Feng, and T. Tse. Metamorphic testing of programs on partial differential equations: A case study. *Proceedings 26th Annual International Computer Software and Applications*, 2002.

[6] T. Y. Chen, J. W. K. Ho, H. Liu, and X. Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC bioinformatics*, 10:24, 2009.

[7] Y. Cheon and G. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. *ECOOP 2002–Object-Oriented Programming*, 2002.

[8] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*, volume 52 of *Lecture Notes in Computer Science*. MIT Press, 1999.

[9] K. Datta, M. Murphy, and V. Volkov. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. *Supercomputing Conference*, 2008.

[10] J. Dongarra and J. D. Croz. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 1990.

[11] M. Feathers. CUnit. http://cppunit.sourceforge.net/, 2000.

[12] I. Foster. *Designing and building parallel programs*. Addison Wesley Publishing Company, 1995.

[13] J. Gil and Y. Shimron. Smaller footprint for Java collections. *ECOOP 2012–Object-Oriented Programming*, 2012.

[14] GNU Compiler Collection. Gcov. http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html.

[15] K. Hasegawa, Y., Iwata, J. I., Tsuji, M., Takahashi, D., Oshiyama, A., Minami and M. Yokokawa. First-principles calculations of electron states of a silicon nanowire with 100,000 atoms on the K computer. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2011.*, 2011.

[16] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal methods in system design*, 2004.

[17] G. Holzmann. The SPIN model checker: Primer and reference manual. 2004.

[18] S. Hozumi, Y. Sato, and S. Chiba. A Tool to Test Calculation Completeness and Orders for HPC Applications (In Japanese). *IPSJ SIGHPC Technical Report, 2013-HPC-140, 1-7*, 2013.

[19] S. Hozumi, Y. Sato, and S. Chiba. HPCUnit: An Unit Testing Framework to Test Completeness of Calculation Divisions During the Optimization Process of Sicentific Applications (In Japanese). *JSSST-SIGPPL Workshop*, 2014.

[20] S. Kamil, K. Datta, S. Williams, and L. Oliker. Implicit and explicit optimizations for stencil computations. *Proceedings of the 2006 workshop on Memory system performance and correctness*, 2006.

[21] G. Kiczales, E. Hilsdale, and J. Hugunin. An overview of AspectJ. *European Conference on Object-Oriented Programming*, 2001.

[22] M. Kim, M. Viswanathan, and S. Kannan. Java-MaC: A run-time assurance approach for Java programs. *Formal methods in system design*, 2004.

[23] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. *ACM Sigarch Computer Architecture News*, 1991.

[24] LDRA Sofware Technology. LDRA Testbed. http://www.ldra.com/testbed.asp.

[25] O. Matlin, E. Lusk, and W. McCune. SPINning parallel systems software. *Model Checking Software*, 2002.

[26] J. McCalpin and D. Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. 1999.

[27] K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 1996.

[28] C. Murphy, G. Kaiser, L. Hu, and L. Wu. Properties of Machine Learning Applications for Use in Metamorphic Testing. In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering ({SEKE08})*, pages 867–872, 2008.

[29] Nvidia, CUDA. Programming guide, 2008.

[30] S. Pervez and G. Gopalakrishnan. Practical model-checking method for verifying correctness of mpi programs. *Parallel Virtual Machine / Message Passing Interface*, 2007.

[31] S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. Formal methods applied to high-performance computing software design : a case study of MPI one-sided communication-based locking. *Software: Practice and Experience*, 40(December 2009):23–43, 2010.

[32] S. Siegel. Efficient verification of halting properties for MPI programs with wildcard receives. *Verification, Model Checking and Abstract Interpretation*, 2005.

[33] S. Siegel and G. Avrunin. Verification of MPI-based software for scientific computation. *International Workshop on Model Checking of Software*, 2004.

[34] S. Siegel and A. Mironova. Using model checking with symbolic execution to verify parallel numerical programs. *International Symposium on Software Testing and Analysis*, 2006.

[35] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '01*, pages 8–8, New York, USA, Nov. 2001. ACM Press.

[36] C.-a. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T. Chen. Metamorphic Testing for Web Services: Framework and a Case Study. *2011 IEEE International Conference on Web Services*, pages 283–290, 2011.

[37] N. Tallent and J. Mellor-Crummey. Scalable fine-grained call path tracing. *Proceedings of the international conference on Supercomputing*, 2011.

[38] The National Aeronautics and Space Administration. FUnit. http://nasarb.rubyforge.org/funit, 2001.

[39] The National Aeronautics and Space Administration. pFUnit. http://opensource.gsfc.nasa.gov/projects/FUNIT, 2005.

[40] S. Vakkalanka, G. Gopalakrishnan, and R. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. *Computer Aided Verification*, 2008.

[41] A. Vo, S. Vakkalanka, and M. DeLisi. Formal verification of practical MPI programs. *ACM Sigplan*, 2009.

[42] E. J. Weyuker. On testing non-testable programs. *Computer Journal*, 25(4):465–470, 1982.

[43] D. Worth, C. Greenough, and L. Chin. A Survey of C and C++ Software Tools for Computational Science. *Science and Technologies Facilities Council*, 2009.

[44] J.-I. I. Yabana, D. Takahashi, A. Oshiyama, T. Boku, K. Shiraishi, S. Okada, and Kazuhiro. A massively-parallel electronic-structure calculations based on real-space density functional theory. *Journal of Computational Physics*, 2010.