COMPASS: A Combined Parallel Subscripted Subscript Benchmark Suite

Akshay Bhosale and Rudolf Eigenmann

University of Delaware, Newark, DE, USA
{akshay,eigenman}@udel.edu

Abstract. This paper describes COMPASS – a suite of irregular benchmarks comprising special patterns referred to as subscripted subscript patterns. When the values of an array appear at the subscript of another array in a *for*-loop, e.g. $a[b[i]]$ with cross-iteration accesses to the host array (array a), such a pattern is referred to as a subscripted subscript pattern. These patterns represent an important class of dynamic, irregular memory access patterns observed in scientific applications and pose a challenge for optimizing compilers. The suite is a collection of subscripted subscript benchmarks from various application domains such as Machine Learning, Linear System Solvers, Adaptive Mesh Refinement, Sorting Algorithms and Sparse Matrix computations. The primary purpose of this suite is to promote the development of advanced compiler analysis and transformation techniques that enable parallelization of the subscripted subscript loops as well as techniques for improving locality and thread synchronization. We present the necessary and sufficient conditions for eventual parallelization of the subscripted subscript loops and discuss techniques described in the literature for determining said conditions. Experimental results show that subscripted subscript loops appear in key program sections and parallelizing them leads to a substantial improvement in the performance of the overall applications.

Keywords: Subscripted Subscript, Parallelization, Benchmarks.

1 Introduction

Benchmark suites are most frequently designed for the scientific study and evaluation of new and existing technologies – hardware and software. Examples of such benchmark suites include the NAS Parallel Benchmarks [3], the SPEC Benchmarks [2, 35], the PARSEC Benchmarks [7], the Rodinia Benchmarks [9] and others. These benchmarks can measure the overall performance as well as quantify the contribution of a certain technology. By contrast, benchmark suites such as the Barcelona OpenMP task suite [13], the EPCC microbenchmark suite [8], the Polybench suite [21], are specialized benchmarks designed for specific purposes and drive new technologies. These benchmarks play an important role in the High Performance Computing (HPC) community when it comes to the development of new compilation techniques for efficient code generation and

measuring performance impact. Often times, when developing a new technique, compiler developers need to scan through a number of benchmarks to collect the ones which fit the requirements for the evaluation of their technique. The chosen benchmarks should be able to demonstrate the impact as well as exhibit the limitations of the proposed techniques. In addition, they should foster further refinement of the current techniques and development of more sophisticated techniques in the future. The Combined Parallel Subscripted Subscript benchmark suite (COMPASS) is a suite of specialized benchmarks for the development and evaluation of compiler techniques for improving the performance of applications with subscripted subscript patterns.

```
1 for (j = 0; j < numPlaced; j++)
2     y[ind[j]] += gamma2[i] * exp(-((xdos[ind[j]]-t)*
3                                   (xdos[ind[j]] - t))/sigma2);
```

Fig. 1: Example Subscripted subscript loop from the EVSL library [18]. Values of array *ind* appear at the subscript of array *y* on line 2.

A number of scientific applications make use of data structures such as arrays, graphs, trees and grids to perform computations on sparse, unstructured input data. The use of such data structures introduces unpredictable memory access patterns which complicates the process of optimizing and parallelizing the application codes. A class of such dynamic, irregular access patterns are subscripted subscript patterns, wherein - an array value appears in the subscript of another array in a *for*-loop, with cross-iteration write accesses to the host array. Figure 1 shows an example subscripted subscript loop. In this work we introduce the various types of subscripted subscript patterns observed in scientific applications and describe the necessary and sufficient conditions for parallelizing loops with such patterns. In addition, we present compiler techniques that have been proposed in the literature for analyzing the input program and verifying the existence of the required conditions for parallelization. Some of the proposed compile-time techniques [19, 20] have been evaluated using benchmark suites such as the *PERFECT* benchmarks [4] that suffer from a number of limitations and are no longer representative of the workloads of today. In other work, run-time techniques [23, 37] have been evaluated using applications from two domains – iterative solvers, and preconditioners for sparse linear systems. This paper presents a subscripted subscript benchmark suite that represents modern workloads and includes applications from a variety of problem domains.

The COMPASS suite [1] is a collection of applications from popular packages and benchmark suites, composed of subscripted subscript loops with increasing levels of complexity. The suite encompasses applications such as sparse matrix-vector multiplication which is at the core of many sparse matrix computations and linear algebra operations, a sorting algorithm that is important in particle method codes in physics, and computations on unstructured meshes – vital part of computational fluid dynamics applications. The chosen applications contain subscripted subscript loops that represent the computationally intensive parts of the application. Furthermore, in these applications the compiler, upon availabil-

ity of the necessary information, stands a chance of parallelizing the subscripted subscript loops. The purpose of this suite is to drive the development of optimizing compilers including such techniques as – advanced symbolic analysis, automatic parallelization, improving data locality, thread synchronization and improving data structure navigation in applications.

In summary, this paper makes the following contributions:

- We present and characterize COMPASS, a benchmark suite for the development of analysis and transformation techniques for improving the performance of applications with subscripted subscripts.
- We present the necessary and sufficient conditions for parallelization of the subscripted subscript loops appearing in the COMPASS benchmarks, and discuss the various analysis techniques proposed in the literature for determining and verifying said conditions.
- We evaluate the performance potential of parallelizing the subscripted subscript loops and discuss their impact on the overall application performance.

2 The Benchmarks

2.1 Overview

The COMPASS suite includes a set of programs that are representative of a class of irregular applications and the subscripted subscript patterns they contain. Table 1 lists the benchmarks in the COMPASS suite. The suite consists of a total of six application codes chosen from a range of application domains. The included benchmarks have either been extracted from publicly available suites, such as the NAS Parallel Benchmarks (NPB) [26] and the SuiteSparse Benchmark Suite [12] or are kernels derived from larger applications, such as the Algebraic Multi-Grid Solver (AMG) [40] and Sparse Factor Analysis (SFA) [28]. The NPB suite consists of benchmarks for solving systems of partial differential equations (PDEs) using various numerical algorithms, solving problems on unstructured grids, sorting a set of integers and solving systems of linear equations. We have included the Integer Sort (IS) and Unstructured Adaptive (UA) benchmarks from NPB in the COMPASS suite. AMG is a parallel Algebraic Multi-Grid solver for linear systems arising from problems on unstructured grids. The COMPASS suite includes the sparse matrix-vector multiplication kernel from an abridged version of AMG – AMGmk. SuiteSparse is a suite of sparse matrix algorithms for solving systems of linear equations. We have included the Supernodal Cholesky factorization application from SuiteSparse in the COMPASS suite. Sparselib++, written in C++, is a library of numerical linear algebra algorithms for efficient sparse matrix computations across various computational platforms. The COMPASS suite includes the C-translated version of the Incomplete Cholesky factorization kernel from Sparselib++. Many scientific applications involve multiplying a sparse matrix with one or more dense matrices. The COMPASS suite includes a Sampled Dense-Dense Matrix Multiplication (SDDMM) benchmark, which performs the element-wise multiplication of an input sparse matrix with two dense

matrices. The SDDMM computation is at the core of many machine learning and data mining applications.

1. **Input Data Sets**: The NPB suite specifies various problem sizes, small test problems to large problems, defined in terms of classes [26]. Of the classes designated for experiments, Class A is the smallest whereas Class F is the largest. The Supernodal Cholesky factorization, SDDMM and Incomplete Cholesky factorization applications require input dense and sparse matrices that satisfy various criteria as defined by the involved algorithm. The dense matrices have been built into the benchmarks. The required sparse matrices can be obtained from the SuiteSparse Matrix collection [17], a large repository of sparse matrices from various application domains. For the AMGmk benchmark, we define a set of three internally generated input sparse matrices in increasing order of their sizes (number of rows $\times$ number of columns).
2. **Verification**: An important characteristic of any benchmark suite is the verification method employed to test the correctness of specific implementations and applied optimizations. All benchmarks in the COMPASS suite include some form of self-validation to test for the correctness of the output, eliminating the need for manual checks.

Benchmark	Origin	Original language	Domain	No. of lines of Code	Input(s)	No. of parallel subscripted subscript loops or loop nests
AMGmk	AMG [40]	C	Sparse Matrix computation	1800	Sparse Matrix, Dense Vector	2
Integer Sort (IS)	NAS Parallel Benchmarks v3.3 [3]	C	Sorting Algorithms	1000	Integers	4
CHOLMOD Supernodal	SuiteSparse [12]	C	Linear System Solvers	1138 (computation)	Sparse Matrix	3
Unstructured Adaptive (UA)	NAS Parallel Benchmarks v3.3 [3]	C	Adaptive Mesh Refinement	8000	Meshes of different sizes	13
SDDMM (C version)	Nisa et al. [28]	CUDA	Machine Learning and Data Mining	250	Sparse and Dense Matrices	1
Incomplete Cholesky (C version)	Sparselib++ [31]	C++	Linear System Solvers	300	Sparse Matrix	1

Table 1: Applications in the COMPASS Benchmark Suite.

2.2 Benchmark Description

1. **AMGmk v1.0**: The AMGmk mini application based on the AMG [40] benchmark, consists of three compute intensive kernels – *Matvec*, *Relax* and *Axpy* of which, the *Matvec* kernel comprises of parallelizable subscripted subscript loops. Kernel *Matvec* performs the multiplication of a sparse matrix

and a dense vector (SpMV) and is the most frequently used kernel in the AMG solve phase [14]. The Compressed Sparse Row (CSR) format is used to store the sparse matrix in memory. Sparse matrices of different sizes used as inputs to the kernel comprise of laplacians. The input vector used in this case is a randomly generated column vector.

2. **Integer Sort (IS)**: The IS benchmark from NPB v3.3 sorts a large number of integers – 2^{23} (CLASS A) to 2^{31} (CLASS D) using bucket sort. The benchmark uses 2^{10} buckets to sort the integers, referred to as *keys*, generated randomly using a portable random number generator. The benchmark requires sorting the *key* array ten times, each time with a slight modification to the input data. Verification is performed after every run as well as after the completion of the ten sorting runs.
3. **CHOLMOD Supernodal**: CHOLMOD [10] derived from the SuiteSparse benchmark suite is a package that provides Cholesky factorization methods for solving large sparse linear systems. Given a sparse, symmetric positive definite matrix A, its Cholesky factorization is represented as $A = LL^T$, where L is a real lower triangular matrix with positive diagonal entries. The supernodal cholesky factorization algorithm improves upon other cholesky factorization methods and delivers substantial parallel performance by implementing and taking advantage of the "supernodal" structure of L. A *supernode* of a sparse cholesky factor L is the set of columns in L that exhibit the same sparsity structure [27].
4. **Unstructured Adaptive (UA)**: The UA benchmark from NPB v3.3 computes the solution of a partial differential equation on an adaptively constructed non-uniform mesh. Meshes are high-dimensional geometric structures formed by an interconnected grid of elements in time and space. An important characteristic of the subscripted subscript patterns in the UA benchmark is that the patterns use multi-dimensional subscript arrays.
5. **Sampled Dense-Dense Matrix Multiplication (SDDMM)**: first performs the multiplication of two dense matrices, and then scales the result by an input sparse matrix. For e.g. $O = ((BA^T) \cdot S)$ where, B and A are the dense matrices and S is the input sparse matrix. The output matrix O in this case is a sparse matrix having the same sparsity pattern as S. SDDMM is the core computation in the formulation of factorization algorithms such as Sparse Factor Analysis (SFA) and Alternating Least Squares (ALS) used in machine learning and data mining applications [28].
6. **Incomplete Cholesky (IChol)**: Iterative methods are commonly used to determine the solution of sparse symmetric linear systems of the form $Ax{=}B$, where A is an $m{\times}n$ sparse matrix, B is an $m{\times}1$ vector and x is an $n{\times}1$ vector to be determined. The solution in this case can be rapidly approximated by using a preconditioner. The Incomplete Cholesky factorization represents an important class of preconditioners for positive definite systems [34]. The factorization considers only the non-zero entries of the input sparse matrix for determining the solution of the linear system.

The COMPASS suite incorporates benchmark codes comprising parallelizable subscripted subscript loops. A number of applications with subscripted subscripts were not included in the COMPASS suite due to the inherent sequential nature of the involved patterns. Yet other applications made use of complex, language-specific data structures and object oriented constructs, limiting the opportunities for compilers to exploit parallelism.

3 Parallelization of Subscripted Subscripts

This section introduces the different types of subscripted subscript patterns observed in the COMPASS benchmark codes and describes the necessary and sufficient conditions for disproving cross-iteration dependences w.r.t. the enclosing loop or loop nest. The subscript expressions involve arrays with either single or multiple levels of indirection. In addition, the subscript arrays can be either one-dimensional or multi-dimensional.

3.1 Patterns with Single Level of Indirection

The simplest form of a subscripted subscript pattern involves an array whose values appear either directly or indirectly at the subscript of another array. Whereas, in some other patterns, the subscript array is part of an expression. These patterns comprise a single level of indirection i.e. only one index array.

(a) *Patterns with One-dimensional subscript arrays*: Values of a subscript array can appear directly or explicitly at the subscript of another array. For the loop shown in Figure 2, values of array *A_rownnz* appear directly at the subscript of array *y_data* on lines 2 and 6. The outer loop on line 1 can be parallelized if no two iterations of the loop access the same element of array *y_data*. In other words, there is no self-output dependence. This is possible only if array *A_rownnz* is injective i.e. $A_rownnz[i] \neq A_rownnz[i']$, $\forall i \neq i'$ and i, $i' \in [0{:}num_rownnz\text{-}1]$. Injectivity of array *A_rownnz* is sufficient for parallelizing the outermost i-loop in this case. Injectivity can appear in the form of strict monotonicity wherein, $A_rownnz[i] < A_rownnz[i']$, $\forall i < i'$. Note that the potential anti-dependence w.r.t. array *y_data* in this case is not a cross-iteration dependence.

```
1  for(i = 0; i < num_rownnz; i++){
2    tempx = y_data[A_rownnz[i]];
3    for(jj=A_i[A_rownnz[i]]; jj<A_i[A_rownnz[i]+1];jj++){
4        tempx +=  A_data[jj] * x_data[A_j[jj]];
5    }
6     y_data[A_rownnz[i]] = tempx;
7  }
```

Fig. 2: Subscripted subscript loop from the AMGmk [40] application. Values of array *A_rownnz* appear explicitly at the subscript of array *y_data* on lines 2 and 6.

The loop shown in Figure 3 is an example of an indirect subscripted subscript pattern. Values of array *col_ptr* appear at the subscript of array p on line 7 via variable *ind*, the loop index of the inner loop on line 2. To parallelize the outermost r-loop, the range of values of array *col_ptr* accessed in an arbitrary iteration $r - [col_ptr[r] : col_ptr[r{+}1]\text{-}1]$ should not overlap with the range accessed in any other iteration of the loop. This condition is satisfied if array *col_ptr* is monotonically increasing, meaning, $col_ptr[r] \leq col_ptr[r']$, $\forall r < r'$ and $r, r' \in [0{:}n_cols$ - 1]. Non-strict monotonicity suffices in this case.

```
1 for (r = 0; r < n_cols; ++r){
2   for (ind = col_ptr[r]; ind<col_ptr[r+1]; ++ind){
3     sm=0;
4     for (t = 0; t < k; ++t){
5       sm += W[r*k + t]*H[row_ind[ind]*k + t];
6     }
7     p[ind] = sm * nnz_val[ind];
8     }
9 }
```

Fig. 3: Subscripted subscript loop from the SDDMM [28] application. Values of array *col_ptr* appear at the subscript of array p on line 7 via the loop index of the inner loop *ind* on line 2.

```
1   for (ie = 0; ie < nelt; ie++) {
2     for (iface = 0; iface < NSIDES; iface++) {
3      for (i = 1; i < LX1-1; i++) {
4        il = idel[ie][iface][i][LX1-1];
5        for (j = 0; j < LX1; j++) {
6          tx[il] = tx[il] + qbnew[ije1][j][i-1]* tmp[ije1][j][col];
7        }
8      }
9     }
10  }
```

Fig. 4: Subscripted subscript loop from the UA application [15] from the NAS Parallel Benchmarks. Values of a 4-dimensional array *idel* on line 4 appear at the subscript of another array *tx* on line 6 via the scalar variable *il*.

(b) *Patterns with Multi-dimensional subscript arrays*: Applications such as UA in the COMPASS suite contain subscripted subscript patterns with multi-dimensional subscript arrays. The host array can be either one-dimensional or multi-dimensional. An example of such a loop pattern is shown in Figure 4. Values of a 4-dimensional array *idel* on line 4 appear at the subscript of another array *tx* on line 6. The first three dimensions of array *idel* − *ie*, *iface* and i are the loop indices of the loops on lines 1, 2 and 3. To parallelize a loop in this case, array *idel* should be injective w.r.t. the dimension indexed by the loop index variable. As an example, to parallelize the outermost *ie*-loop, array *idel* should possess injectivity w.r.t. the dimension indexed by

ie or dimension at the first position. If this condition is satisfied, no cross-iteration output dependences will exist w.r.t array *tx* and the outermost loop can then be parallelized.

(c) *Patterns with Subscript Expressions*: Some benchmarks comprise subscripted subscript patterns wherein, the subscript array (say array B) is part of an expression of the form $\alpha * B[i] + \beta * j$, where i, j are index variables of the enclosing loops and α and β are loop invariant values. For the loop shown in Figure 5, values of array *mt_to_id_old* appear at the subscript of array *ref_front_id* on line 8. Additionally, the subscript array *front* is part of two expressions, one on line 3 and another one on line 5. The values of these two expressions appear at the subscript of array *mt_to_id* on line 7. The outermost loop can be parallelized if array *mt_to_id_old* is injective and the expressions on lines 3 and 5 produce sets of values which are not only injective but also mutually exclusive.

```
1  for(miel = 0; miel < nelt; miel++) {
2     if (ich[mt_to_id_old[miel]] == 4) {
3        mielnew = miel + (front[miel]-1)*7;
4     } else {
5        mielnew = miel + front[miel]*7;
6     }
7      mt_to_id[mielnew] = mt_to_id_old[miel];
8      ref_front_id[mt_to_id_old[miel]] = nelt+ntemp;
9  }
```

Fig. 5: Subscripted subscript loop from the UA benchmark [15] from the NAS Parallel Benchmarks. Values of subscript array *mt_to_id_old* appear explicitly at the subscript of array *ref_front_id* on line 8. In addition, values of two subscript expressions can appear at the subscript of array *mt_to_id* on line 7. The two expressions, one on line 3 and another one on line 5 produce sets of values which are injective and mutually exclusive.

3.2 Patterns with Multiple Levels of Indirection

Some benchmark applications involve loop nests with arrays consisting of multiple levels of indirection in their subscripts, such as $A[B[C[i]]]$. An example of such a loop pattern is shown in Figure 6 wherein, values of subscript arrays *colPtr* and *rowIdx* appear at the subscript of array *val* on line 11 via k, the loop index of the inner loop on line 8. Array *val* is defined and used in statements on lines 2, 5 and 11. The outermost i-loop in this case cannot be parallelized due to cross-iteration flow and anti-dependences w.r.t. array *val*. However, the inner m-loop on line 7 can be parallelized if – (a) subscript array *colPtr* is monotonic, (b) array *rowIdx* is strictly monotonic i.e. $rowIdx[colPtr[i]{+}1] < rowIdx[colPtr[i{+}1]{-}1]$, $\forall i \in [0{:}n{-}1]$ and (3) $rowIdx[m] \geq i{+}1$. If the aforementioned constraints are satisfied, the section of array *val* that is read on line 11 will never overlap with the section of the array which is modified for all iterations of the m-loop on line 7. In addition, no two iterations of the loop will modify the same element of array *val*, eliminating any possible output dependence.

```
1  for(i = 0; i < n; i++){
2    val[colPtr[i]] = sqrt(val[colPtr[i]]);
3
4    for(m=colPtr[i]+1; m<colPtr[i+1]; m++){
5      val[m] = val[m]/val[colPtr[i]];
6    }
7   for(m = colPtr[i]+1; m<colPtr[i+1]; m++) {
8    for(k = colPtr[rowIdx[m]]; k<colPtr[rowIdx[m]+1]; k++){
9     for(l = m; l<colPtr[i+1]; l++){
10      if(rowIdx[l]==rowIdx[k]&&rowIdx[l+1]<=rowIdx[k]){
11         val[k] -= val[m]* val[l];
12 }}}}}
```

Fig. 6: Subscripted subscript loop from the Incomplete Cholesky implementation from the SparseLib++ library [23, 31]. Values of two arrays *colPtr* and *rowIdx* appear at the subscript of array *val* on line 11 via *k*, the loop index of the inner loop on line 8. The *m*-loop on line 7 can be parallelized if array *colPtr* is monotonic, array *rowIdx* is strictly monotonic and $rowIdx[m] \geq i{+}1, \forall i \in [0{:}n\text{-}1]$.

4 Index Array Analysis Techniques

We review the various index array analysis techniques presented in the literature for determining and verifying the necessary and sufficient conditions for parallelization of subscripted subscripts discussed in the previous section. The techniques fall into one of three categories – compile-time, run-time or hybrid.

4.1 Compile-time Analysis Techniques

Compile-time analysis techniques rely on static information about index arrays such as a property or range of possible values, which is then used to test for the presence or absence of dependences in the subscripted subscript loops where the index arrays appear. This information is either derived from the application code itself or is provided by the user.

Demand-driven Array Property Analysis: Lin and Padua [19,20] presented a demand-driven interprocedural query propagation technique for analyzing index arrays and determining index array properties. Their technique analyzes program patterns that define index arrays. The technique has three main parts – (i) *The QueryGenerator*, which generates a query when a test needs to verify whether an index array has a property at a certain point (ii) *The QuerySolver* which traverses the program in the reverse order of the control flow to verify the query and (iii) *The PropertyChecker* to get the GEN and $KILL$ information about the arrays and check for the property. If a property is available, the *QuerySolver* returns *true* to the *QueryGenerator*, otherwise it returns *false*. Their technique can determine properties such as injectivity, closed form value and closed form bound by using pattern matching. For example, their technique can determine injectivity of an index array only when the array is defined in an

index gathering loop (loop that assigns values of the index variable to the array). When evaluated on a set of 4 benchmarks (3 from the *PERFECT* suite [4] and 1 from NCSA), their technique could determine the properties – closed form value and closed form bound for the index arrays, and led to significant performance improvement in 2 out of the 4 benchmarks evaluated.

Using User-defined Assertions: Properties of index arrays can be expressed as user-defined assertions which can be tested at run-time [22]. Mohammadi et al. [23] presented a technique that leverages constraint-based data dependence analysis to find parallelism in subscripted subscript loops. The dependence relations for every loop in an input application code are extracted using the CHILL polyhedral compiler framework [38]. The extracted loop dependences and the corresponding user-defined assertions are provided as inputs to a Z3 SMT solver [25] which tests if the constraints are satisfiable or unsatisfiable. If all of the dependences for a loop are unsatisfiable under the specified constraints, the loop can be parallelized. When applied to the computationally intensive loop from the Incomplete Cholesky factorization application shown in Figure 6, their technique shows a significant improvement in the overall application performance.

Recurrence Recognition: Bhosale and Eigenmann [5, 6] proposed an algorithm based on *symbolic range aggregation* that can determine subscript array properties by analyzing loops that initialize and modify the content of the subscript array. In most cases, program statements that create a property such as monotonicity, comprise *recurrence relationships* where, the next value in the sequence is computed from the previous value plus a positive or non-negative increment. Array recurrences are of particular interest. Their technique can automatically parallelize subscripted subscript loops and improve the performance of the AMGmk, CHOLMOD Supernodal, SDDMM and UA benchmarks from the COMPASS suite.

4.2 Run-time Analysis Techniques

In some benchmarks, the subscript array is initialized and modified in loop patterns that are too complex to analyze at compile-time. Whereas, in some other applications, either the value assigned to the subscript array is input-dependent or the subscript array itself is read from an input file. In such cases where, compile-time analysis is conservative and assumes dependences due to insufficient information about the subscript arrays, run-time techniques can be used to verify the necessary and sufficient conditions for parallelization.

Inspector-Executor Approach: The inspector-executor technique was first introduced by Saltz et al. [33] and used extensively in the sparse polyhedral framework by Strout et al. [36, 37, 39]. The technique implements an inspector code which traverses the index array at run-time to determine the data access

patterns. In doing so, the relevant index array properties can be realized. This information can then be used by the executor code, which is a transformed version of the source code loop structures, to execute the computation in an efficient manner. A challenge in this case is the complexity of the generated inspectors. In applications such as Incomplete Cholesky factorization, the cost of inspection is higher than the gains realized by parallelization of the computational kernel. Mohammadi et. al. [24] proposed techniques for simplifying dependence constraints and generating efficient inspectors. Their techniques were successful in reducing the complexity of the originally generated inspectors. However, in three of the seven benchmarks evaluated (the three benchmarks being iterative solvers), the inspector overhead is still significantly greater than the execution time of the kernel. For these benchmarks, the parallel executor has to be run multiple times to realize the performance gains over the serial code. Due to this drawback, this technique is not well suited for computational kernels with smaller workloads.

Speculative Execution: The distribution of the original loop into an inspector and executor loop is often disadvantageous. As mentioned earlier, in many cases, the generated inspector can be both computationally expensive and with side-effects. In addition, extracting the appropriate inspector automatically can be a challenge in some applications. Rauchwerger et al. [11,32] presented a framework for speculatively executing a sequential loop as parallel and applying a fully parallel data dependence test to determine if the loop had any cross-iteration dependences. If the run-time test fails, then the loop is re-executed sequentially. Even with the possibility of a heavy penalty if the dependence test fails, the authors argue that speculative execution followed by run-time analysis is the only viable solution for many subscripted subscript loops.

4.3 Hybrid Analysis Techniques

Hybrid Analysis techniques presented by Oancea and Rauchwerger [29, 30] aim to perform both compile-time and run-time analysis with a time complexity that is as close as possible to that of compile-time analysis. Their techniques are capable of determining monotonicity of array references. A summary of the array accesses in a loop is constructed using interprocedural dataflow analysis, expressed in the form of symbolic sets. The aggregated sets are used by the run-time analysis to test for the presence of monotonicity. Their techniques make use of pattern matching at the level of the symbolic sets to determine where to insert the monotonicity tests. Their techniques could determine monotonicity of subscript arrays in the TRFD and DYFESM benchmarks from the *PERFECT* [4] benchmark suite.

5 Evaluation

We evaluate the performance potential of parallelizing the subscripted subscript loops appearing in the COMPASS applications. Our results show that sub-

scripted subscript loops represent major performance bottlenecks in the application codes; parallelizing them leads to a significant improvement in the overall application performance.

5.1 Experimental Setup

The evaluation is performed in two steps – (1) profiling the application codes to determine the total execution time of the subscripted subscript loops relative to the application execution time and (2) determining the performance improvement of the application codes after hand parallelizing the subscripted subscript loops, showing the performance potential of the parallel loops. Table 2 lists the problem classes and sparse matrices used as inputs for our experiments. For the Integer Sort and Unstructured Adaptive benchmarks, we used the problem classes A, B and C defined in the NPB suite as input datasets. For the AMGmk benchmark we used three internally generated sparse matrices as inputs to the application code. For the CHOLMOD Supernodal, SDDMM and Incomplete Cholesky benchmarks, we used large sparse matrices from the SuiteSparse matrix collection [17] as inputs. The selected input matrices satisfy the criteria related to the size, shape, structure and type of entries in the matrix, as defined by the involved algorithm.

We used the Intel Vtune profiler [16] with user-mode sampling enabled to collect the profiling results for the application codes. The default scheduling scheme for the parallel applications was set to static scheduling except in the Incomplete Cholesky application. For this application, the scheduling scheme was set to cyclic (static scheduling with a chunk size of 1), the reason being that cyclic scheduling outperforms all other scheduling schemes across all input matrices and varying number of cores. The execution times were recorded on a compute node with a 20-core Intel Xeon Gold 6230 processors in a dual socket configuration, with a processor base frequency of 2.095 GHz, 27.5MB cache and we used upto 16GB of DDR4 memory. The application codes were compiled using GCC v4.8.5 with the -O3 optimization flag enabled on CentOS v7.4.1708 and we report the mean of 5 application runs. We observed a maximum run-to-run variation of 2% and we used one thread per core.

5.2 Results

Table 2 shows the profiling results for the serial COMPASS benchmarks for the input classes and sparse matrices. The table also shows the total memory consumed by the application codes. We report the single core execution time of the parallelizable subscripted subscript loops relative to the total application execution time. The SDDMM, Incomplete Cholesky and CHOLMOD Supernodal applications spend substantial time in I/O operations such as reading in the input sparse matrix, in addition to the actual computation. For these applications, we report the execution time of the actual computation. From Table 2 it can be

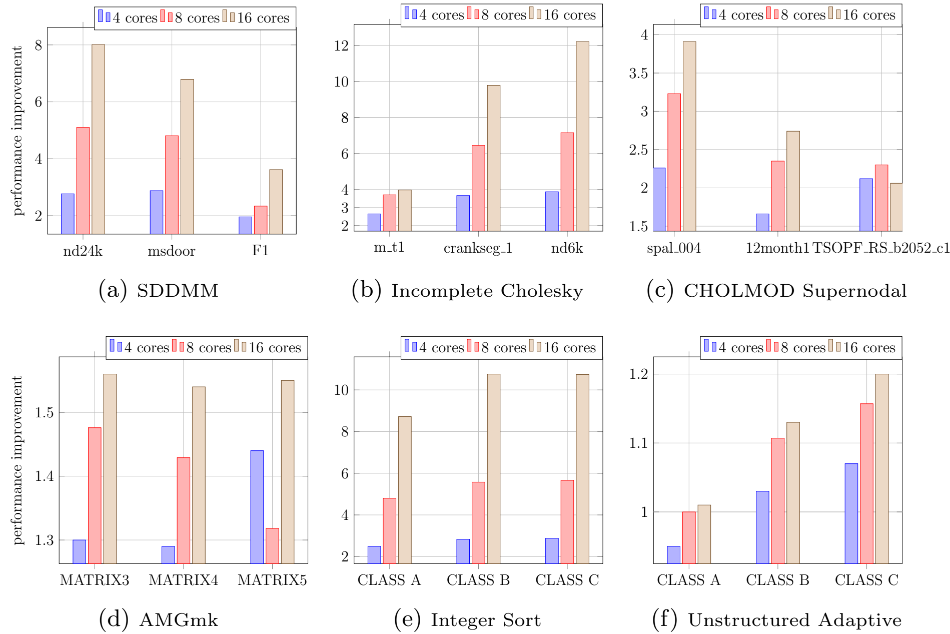
inferred that subscripted subscript loops represent a major performance bottleneck in the COMPASS benchmarks taking anywhere between 15.1% to 98.34% of the total application execution time.

Benchmark	Input Classes/Matrices	Total CPU time taken by the application (s)	Time taken by the parallelizable Subscripted Subscript loops (s/%)	Total Memory consumption
AMGmk	MATRIX3 (5M×5M, 6.94M nnz)	7.61 s	4.42 s /58.1%	227.3 MB
	MATRIX4 (8.64M×8.64M, 12M nnz)	14.030 s	8.01 s /58.01%	393 MB
	MATRIX5 (16.8M×16.8M, 23.49M nnz)	27.15 s	15.731s /57.94%	768 MB
Integer Sort	CLASS A	1.36 s	0.5 s /36.76%	4.1 KB
	CLASS B	5.49 s	2.19 s /39.98%	4.1 KB
	CLASS C	22.26 s	9.16 s /41.15%	4.1 KB
CHOLMOD Supernodal	spal_004 (10.2k×321.6k, 46.1M nnz)	11.11 s	10.69 s /96.21%	4.7 GB
	12month1 (12.4k×872.6k, 22.6M nnz)	12.28 s	11.61 s /94.54%	4.8 GB
	TSOPF_RS_b2052_c1 (25.6k×25.6k, 6.76M nnz)	69.66 s	58.46 s /83.92%	4.8 GB
Unstructured Adaptive	CLASS A	14.23 s	2.15 s /15.1%	32.8 KB
	CLASS B	75.93 s	14.11 s /18.58%	32.8 KB
	CLASS C	351.47 s	62.201 s /17.69%	32.8 KB
SDDMM	nd24k (72k×72k, 14.39M nnz)	1.25 s	1.19 s /95.2%	518 MB
	msdoor (415.8k×415.8k, 10.3M nnz)	0.89 s	0.86 s /96.62%	954.6 MB
	F1 (343.79k×343.79k, 13.5M nnz)	1.21 s	1.19 s /98.34%	930.6 MB
Incomplete Cholesky	m_t1 (97.5k×97.5k, 4.92M nnz)	6.33 s	4.91 s /77.56%	1.6 GB
	crankseg_1 (52.8k×52.8k, 5.33M nnz)	27.27 s	25.66 s /94.1%	1.7 GB
	nd6k (18k×18k, 3.45M nnz)	36.51 s	35.05 s /96%	1.1 GB

Table 2: Profiling results for the serial COMPASS benchmarks. The measurements were gathered using the Intel VTune profiler [16] on a single core. For input sparse matrices, we have mentioned the size of the matrix (number of rows × number of columns) and the number of non-zeros (nnz) in the matrix.

Figure 7 shows the performance results of the COMPASS application codes for the various inputs. We set the execution time of the fully serial code as the baseline for our experiments. Performance improvement is the comparison of the execution time of the application codes after hand parallelizing the key subscripted subscript loops against the baseline. We report the performance improvement on 4, 8 and 16 cores. For the SDDMM, Integer Sort and Incomplete Cholesky Factorization applications, we obtained a maximum performance improvement of 8.01×, 10.76× and 12.22× respectively for the input classes/sparse matrices. In addition, the performance scales with increase in the number of cores.

In contrast, for the CHOLMOD Supernodal application, performance scales for all the input sparse matrices except TSOPF_RS_b2052_c1. For this matrix, the application performance on 16 cores is about 11.56% less than the performance on 8 cores. Similar result is obtained for the AMGmk benchmark with input matrix – MATRIX5 wherein, the performance of the application on 8 cores is about 9.25% less than the performance on 4 cores. The reason for this discrepancy is in part due to load imbalance issues, resulting from the sparsity pattern i.e. the distribution of nonzeros across the columns (or rows) of the input sparse matrices.



(a) SDDMM (b) Incomplete Cholesky (c) CHOLMOD Supernodal

(d) AMGmk (e) Integer Sort (f) Unstructured Adaptive

Fig. 7: Improvement in the performance of the application codes observed after parallelizing the subscripted subscript loops. The baseline here is the execution time of the fully serial code.

For the Unstructured Adaptive benchmark, we obtained a maximum performance improvement of 1.2× and the performance scales with increase in the number of cores for each of the input classes. In this benchmark, the parallelizable subscripted subscript loops take between 15.1% to 18.58% of the total application execution time. Therefore, parallelizing the subscripted subscript loops leads to a maximum improvement in the performance of the application code by about 7.1% for input Class A, 15.7% for Class B and 20% for Class C over the serial baseline. A key evaluation result here is that the parallel execution of subscripted subscript loops yields significant improvement in the performance of the overall applications.

6 Conclusions

We presented the COMPASS (*Combined Parallel Subscripted Subscript*) benchmark suite, a collection of irregular applications with subscripted subscripts. The suite can aid programmers in the development and evaluation of advanced compiler techniques aimed at exploiting parallelism and data locality, in order to improve the performance of such applications. In addition, we presented a detailed analysis of the subscripted subscript patterns appearing in the benchmark codes, and identified the necessary and sufficient conditions for parallelizing loops with such patterns. Our results show that subscripted subscript loops represent a major performance bottleneck in scientific applications. Parallelizing these loops led to good speedups for most applications, though in some cases we observed a drop in the application performance due to load imbalance issues. The COMPASS suite can also foster the development of analysis techniques for better synchronization and scheduling of loop iterations for further performance improvements.

References

1. Akshay Bhosale: The COMPASS Benchmark Suite (5 2023), https://github.com/akshay9594/COMPASS/
2. Aslot, V., Domeika, M., Eigenmann, R., Gaertner, G., Jones, W.B., Parady, B.: SPEComp: A new benchmark suite for measuring parallel computer performance. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). vol. 2104 (2001). https://doi.org/10.1007/3-540-44587-0_1
3. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The nas parallel benchmarks. International Journal of High Performance Computing Applications **5**(3) (1991). https://doi.org/10.1177/109434209100500306
4. Berry, M., Chen, D., Koss, P., Kuck, D., lo, S., Pang, Y., Pointer, L., Roloff, R., Sameh, A., Clementi, E., Chin, S., Schneider, D., Fox, G., Messina, P., Walker, D., Hsiung, C., Schwarzmeier, J., Lue, K., Orszag, S., Seidl, F., Johnson, O., Goodrum, R., Martin, J.: The perfect club benchmarks: Effective performance evaluation of

supercomputers. International Journal of High Performance Computing Applications **3**(3) (1989). https://doi.org/10.1177/109434208900300302
5. Bhosale, A., Eigenmann, R.: Compile-time Parallelization of Subscripted Subscript Patterns. In: Proceedings - 2020 IEEE 34th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020 (2020). https://doi.org/10.1109/IPDPSW50202.2020.00065
6. Bhosale, A., Eigenmann, R.: On the automatic parallelization of subscripted subscript patterns using array property analysis. In: Proceedings of the International Conference on Supercomputing (2021). https://doi.org/10.1145/3447818.3460424
7. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite. In: Proceedings of the 17th international conference on Parallel architectures and compilation techniques. pp. 72–81. ACM, New York, NY, USA (10 2008). https://doi.org/10.1145/1454115.1454128, https://dl.acm.org/doi/10.1145/1454115.1454128
8. Bull, J.M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In Proceedings of First European Workshop on OpenMP (1999)
9. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009 (2009). https://doi.org/10.1109/IISWC.2009.5306797
10. Chen, Y., Davis, T.A., Hager, W.W., Rajamanickam, S.: Algorithm 887: CHOLMOD, supernodal sparse cholesky factorization and update/-downdate. ACM Transactions on Mathematical Software **35**(3) (2008). https://doi.org/10.1145/1391989.1391995
11. Dang, F., Yu, H., Rauchwerger, L.: The R-LRPD test: Speculative parallelization of partially parallel loops. In: Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2002 (2002). https://doi.org/10.1109/IPDPS.2002.1015493
12. Davis, T.A., Rajamanickam, S., Sid-Lakhdar, W.M.: A survey of direct methods for sparse linear systems (2016). https://doi.org/10.1017/S0962492916000076
13. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguadé, E.: Barcelona openMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openMP. In: Proceedings of the International Conference on Parallel Processing (2009). https://doi.org/10.1109/ICPP.2009.64
14. Falgout, R.D., Li, R., Sjögreen, B., Wang, L., Yang, U.M.: Porting hypre to heterogeneous computer architectures: Strategies and experiences. Parallel Computing **108** (2021). https://doi.org/10.1016/j.parco.2021.102840
15. Feng, H., Van Wijngaart, R.D., Biswas, R.: Unstructured adaptive meshes: Bad for your memory? In: Applied Numerical Mathematics. vol. 52 (2005). https://doi.org/10.1016/j.apnum.2004.08.029
16. Intel: Intel VTune profiler (5 2023), https://tinyurl.com/mrx7n854
17. Kolodziej, S., Aznaveh, M., Bullock, M., David, J., Davis, T., Henderson, M., Hu, Y., Sandstrom, R.: The SuiteSparse Matrix Collection Website Interface. Journal of Open Source Software **4**(35) (2019). https://doi.org/10.21105/joss.01244
18. Li, R., Xi, Y., Erlandson, L., Saad, Y.: The Eigenvalues Slicing Library (EVSL): Algorithms, implementation, and software. SIAM Journal on Scientific Computing **41**(4) (2019). https://doi.org/10.1137/18M1170935
19. Lin, Y., Padua, D.: Analysis of irregular single-indexed array accesses and its applications in compiler optimizations. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). vol. 1781 (2000). https://doi.org/10.1007/3-540-46423-9_14

20. Lin, Y., Padua, D.: Demand-driven interprocedural array property analysis. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **1863** (2000). https://doi.org/10.1007/3-540-44905-1_19
21. Louis-Noel Pouchet: PolyBench/C (6 2023), https://web.cse.ohio-state.edu/ pouchet.2/software/polybench/
22. McKinley, K.: Dependence analysis of arrays subscripted by index arrays. Tech. rep., Rice University, Houston (1991). https://doi.org/http://www.crpc.cs.rice.edu/softlib/TR_scans/CRPC-TR91153throughTR91187/CRPC-TR91163.PDF
23. Mohammadi, M.S., Cheshmi, K., Dehnavi, M.M., Venkat, A., Yuki, T., Strout, M.M.: Extending index-array properties for data dependence analysis. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). vol. 11882 LNCS (2019). https://doi.org/10.1007/978-3-030-34627-0_7
24. Mohammadi, M.S., Davis, E.C., Nandy, P., Yuki, T., Hall, M., Olschanowsky, C., Strout, M.M., Cheshmi, K., Dehnavi, M.M., Venkat, A.: Sparse computation data dependence simplification for efficient compiler-generated inspectors. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2019). https://doi.org/10.1145/3314221.3314646
25. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver Tools and Algorithms for the Construction and Analysis of Systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer Berlin Heidelberg (2008), http://www.springerlink.com/content/60hx121083823548
26. NASA: NAS Parallel Benchmarks (2023), https://www.nas.nasa.gov/software/npb.html
27. Ng, E., Peyton, B.W.: A Supernodal Cholesky Factorization Algorithm for Shared-Memory Multiprocessors. SIAM Journal on Scientific Computing **14**(4) (1993). https://doi.org/10.1137/0914048
28. Nisa, I., Sukumaran-Rajam, A., Kurt, S.E., Hong, C., Sadayappan, P.: Sampled Dense Matrix Multiplication for High-Performance Machine Learning. In: Proceedings - 25th IEEE International Conference on High Performance Computing, HiPC 2018 (2019). https://doi.org/10.1109/HiPC.2018.00013
29. Oancea, C.E., Rauchwerger, L.: Logical inference techniques for loop parallelization. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2012). https://doi.org/10.1145/2254064.2254124
30. Oancea, C.E., Rauchwerger, L.: A hybrid approach to proving memory reference monotonicity. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). vol. 7146 LNCS (2013). https://doi.org/10.1007/978-3-642-36036-7_5
31. Pozo, R., Remington, K., Lumsdaine, A.: Sparselib++ v. 1.5 Sparse Matrix Class Library Reference Guide (2 1996)
32. Rauchwerger, L., Padua, D.A.: The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. IEEE Transactions on Parallel and Distributed Systems **10**(2) (1999). https://doi.org/10.1109/71.752782
33. Saltz Joel, M.R.: The preprocessed doacross loop. Tech. rep., NASA, Hampto, VA (1990), https://apps.dtic.mil/sti/pdfs/ADA227203.pdf
34. Scott, J., Tůma, M.: On positive semidefinite modification schemes for incomplete cholesky factorization. SIAM Journal on Scientific Computing **36**(2) (2014). https://doi.org/10.1137/130917582

35. SPEC: SPEC CPU 2017 Benchmarks (2023), https://www.spec.org/cpu2017/
36. Strout, M.M., Hall, M., Olschanowsky, C.: The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. Proceedings of the IEEE **106**(11) (2018). https://doi.org/10.1109/JPROC.2018.2857721
37. Strout, M.M., LaMielle, A., Carter, L., Ferrante, J., Kreaseck, B., Olschanowsky, C.: An approach for code generation in the Sparse Polyhedral Framework. Parallel Computing **53** (2016). https://doi.org/10.1016/j.parco.2016.02.004
38. Tiwari, A., Chen, C., Chame, J., Hall, M., Hollingsworth, J.K.: A scalable auto-tuning framework for compiler optimization. In: IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium (2009). https://doi.org/10.1109/IPDPS.2009.5161054
39. Venkat, A., Mohammadi, M.S., Park, J., Rong, H., Barik, R., Strout, M.M., Hall, M.: Automating Wavefront Parallelization for Sparse Matrix Computations. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC. vol. 0 (2016). https://doi.org/10.1109/SC.2016.40
40. Yang, U.M.: Parallel algebraic multigrid methods - High performance preconditioners. Lecture Notes in Computational Science and Engineering **51** (2006). https://doi.org/10.1007/3-540-31619-1_6