ParSolGen - a language and automated parallel programs construction tool for numerical algorithms

1st Nikolai Beliaev
Munchen, Germany
bl0ckzer02@gmail.com, ORDCID: 0000-0002-8455-0995

***Abstract*—The paper is devoted to the problem of the automated construction of a highly efficient parallel program from a given numerical algorithm description. ParSolGen (Parallel Solvers Generator) - a language and parallel program construction tool is presented. ParSolGen is able to automatically construct a highly efficient parallel program from a given numerical algorithm description written in ParSolGen language. The key feature of ParSolGen is its ability to integrate various domain-specific techniques and algorithms for parallel program construction into a single tool. Instead of using general-purpose task-based tun-time systems or performing sophisticated information dependencies analysis, ParSolGen compiler classifies the input numerical algorithm description and selects the integrated domain-specific parallel program construction module based on to the class to which the numerical algorithm belongs. This allows employing domain-specific manual parallel programming techniques in the process of the automated parallel program construction. This approach improves the performance of the automatically generated parallel programs, as confirmed by performance tests. ParSolGen showed the performance of the automatically constructed parallel programs that is comparable to that of the popular library implementation of the same algorithms.**

***Index Terms*—Automated parallel programming, DAG, task graph, task-based parallelism, task-based run-time**

I. Introduction

The growing complexity and heterogeneity of modern supercomputers often complicates the manual parallel implementation of numerical algorithms. In order to achieve high efficiency of numerical parallel programs, developers are often have to address system parallel programming challenges such as load balancing and tasks scheduling. Otherwise, costly computational resources are wasted. However, system parallel programming is often outside of the numerical parallel program developer's scope. This raises the problem of the automation of the numerical parallel programming. In order to address this problem a variety of automated parallel programming systems, languages and tools have been actively developed over recent decades. Tools that support the automated construction of a parallel program for a distributed-memory parallel computer from a given numerical algorithm description are of particular interest. These tools can be categorized into two main classes. General-purpose tools are able to automatically construct a parallel program from any given numerical algorithm description. The price for such versatility is often relatively poor performance of the automatically constructed programs. On the contrary, specialized tools can only handle a limited class of numerical algorithms (for example, [1] is designed for dense linear algebra domain). Despite that, such specialized tools are able to employ domain-specific optimizations and parallel programming techniques in the process of the parallel program construction. This often improves the constructed parallel programs performance significantly. It is clear that various domain-specific parallel programming techniques can be adapted and integrated into a single general-purpose parallel program construction tool. Such accumulation potentially enables the automated construction of highly efficient parallel programs with the versatility of such tool approaching that of general-purpose automated parallel program construction tools.

In this paper ParSolGen [2], [3] (Parallel Solvers Generator), an automated parallel programs construction tool, is presented. ParSolGen constructs a highly efficient parallel program from a given numerical algorithm description written in ParSolGen language. Unlike other general-purpose and domain-specific tools, systems and languages, ParSolGen supports the accumulation of various domain-specific parallel program construction algorithms and techniques. Instead of performing sophisticated information dependencies analysis of the input numerical algorithm description or employing general-purpose parallel program construction techniques and algorithms, ParSolGen classifies the input numerical algorithm description. Based on the classification results, ParSolGen selects the domain-specific module best suited for constructing a parallel program for the identified class of numerical algorithms. This allows ParSolGen to leverage the advantages of domain-specific automated parallel programming techniques while approaching the versatility of general-purpose tools.

The rest of the paper is organized as follows. Section 2 provides related work overview. Section 3 describes the numerical algorithm representation used in ParSolGen. Section 4 gives illustrative examples of ParSolGen language. Section 5 describes the details of ParSolGen internals. The next sections provides a detailed explanation of the automated parallel program construction algorithms and approaches implemented in ParSolGen. This section is followed by performance evaluation results discussion and conclusion section.

II. Related work

Consider three categories of the existing automated parallel program construction tools. General-purpose run-time systems represent the input numerical algorithm description as a DAG (Directed Acyclic Graph) of tasks with each being associated with a user-provided sequential implementation (*kernel*). The DAG is fed into the input of the run-time system which executes it on a distributed memory parallel computer. Legion [4], Charm++ [5], [6], QUARK [7] and StarPU [8] can serve as illustrative examples of such approach. Legion uses regions (named sets of objects) concept to express data locality. Regions describe the organization of data and information dependencies between tasks. Charm++ describes a numerical algorithm as a DAG of so called chares (computational tasks). Chares can communicate to each other by sending and receiving asynchronous messages. QUARK and StarPU are both general-purpose task-based runtime systems that allow submitting tasks through the API of run-time system. Both systems build a tasks DAG automatically from the set of tasks provided by the user and then execute it. Considered general-purpose run-time systems are able to handle any given DAG representation of a numerical algorithm. Despite that, the tasks DAG representation often is not sufficient for the further classification of a numerical algorithm. This limits the ability to integrate domain-specific parallel program construction modules into such systems.

At the opposite of the spectrum various domain-specific compilers and run-time systems have been actively developed. Halide [9], ParSEC [10] and DPLASMA [1] can serve as examples of such tools. Halide provides a DSL(Domain-Specific Language) that allows expressing schedules of stages of an image processing pipeline. Helide compiler is able to automatically construct a parallel implementation of a given image processing pipeline description. DPLASMA [1] allows the automated execution of a task-based DAG representation of a dense linear algebra algorithm. It employs domain-specific DAG execution techniques which are implemented in the underlying DAGuE [11] engine. The same applies to ParSSEC run-time system, which executes a linear-algebra algorithm represented as a PTG (Parameterized Task Graph) of tasks in a distributed manner. Such tools are not universal and can handle only a restricted class of numerical algorithms. However, they employ domain-specific parallel program construction techniques and optimizations that enhance the performance of the constructed parallel programs. These domain-specific techniques are valuable, as they can be integrated into a general-purpose tool to improve the automatically constructed parallel programs performance.

Automated parallel program construction tools that combine a general-purpose high-level DSL compiler with a general-purpose runtime system are of particular interest, as they provide powerful means to integrate domain-specific parallel program construction algorithms and techniques. Regent [12], [13] and LuNA [14] are illustrative examples of such systems. Regent compiler optimizes the input numerical algorithm description written in Regent DSL and generates an executable representation of it. This executive representation can then be executed by Legion run-time system. Similarly, LuNA consists of a DSL compiler and a general purpose run-time system. The LuNA compiler analyzes the input algorithm description written in LuNA language and generates an executive representation of the input numerical algorithm which is then executed by general-purpose task-based LuNA run-time system.

ParSolGen itself is a successor of LuNA project. Unlike LuNA or Regent, ParSolGen is designed to accumulate various domain-specific parallel program construction algorithms and techniques.

III. Numerical algorithm representation

In ParSolGen numerical algorithm is represented as a tuple of three sets (potentially infinite): a set of single execution operations, a set of single-assignment variables and a set of arrays of variables. Variables store arbitrary data (for example: matrix blocks, vector blocks, single values, etc.). Each variable is assigned its type and an integer (version of the variable). Two different versions of a variable are considered as two independent variables by ParSolGen compiler. However, the values of these two variable versions can be stored within the same memory location (which is decided by ParSolGen compiler). Variables can be aggregated into n-dimensional arrays in a manner similar to most procedural programming languages. Each operation is associated with a set of input variables, a set of output variables and a sequential subroutine (*kernel*) provided by the developer of the numerical algorithm and written in C or C++ language. An operation computes the values of the values of its output variables from the given values of the input variables. This is done by executing the associated kernel. An operation can be executed when all its input variable values are computed (in particular if the set of the input variables is empty). After an operation is executed the values of its output variables are considered computed and thus other operations can be executed. The input numerical algorithm is considered executed when all operations are executed. This representation of a numerical algorithm can be illustrated as a DAG (Directed Acyclic Graph) of operations and variables with the arcs of the DAG representing the information dependencies between operations.

IV. ParSolGen language

The ParSolGen language provides convenient means of describing graph representation of numerical algorithms. The following statements are supported by the ParSolGen language: variable, array, subroutine and operation declaration statements, for and while loop statements and conditional statements. Consider the following example of the numerical algorithm for summing two vectors described in ParSolGen.

```
import init_vec(in int i, in int blockn,
        out Array<double> vec[blockn]);
import vec_sum(in int blockn,
        in Array<double> x[blockn],
        in Array<double> x1[blockn],
        out Array<double> y[blockn]);


...


int blockn = 10;
int nb = 5;

Array<double[blockn]> a[nb];
Array<double[blockn]> b[nb];
Array<double[blockn]> c[nb];

for (int i = 0; i < nb; ++i)
{
        init_vec(i, blockn, a[i]);
        init_vec(i, blockn, b[i]);
}

for (int i = 0; i < nb; ++i)
{
        vec_sum(blockn, a[i], b[i], c[i]);
}
```

In the example code snippet three kernels are imported: *init_vec*, *print_vec* and *vec_sum*. This is done by using *import* statement which links the name of a kernel with its sequential implementation written in C++ and stored in a separate file. For each imported kernel a set of its arguments is specified to allow binding the corresponding C++ subroutine parameters to the actual ParSolGen operation input and output variable values. For each kernel argument declaration it is specified whether the argument is input or output along with its type. In the example it is assumed that *init_vec* implements vector block initialization, *vec_sum* sums two vector blocks.

Next, two integer variables ($blockn = 10$ and $nb = 5$) are declared. Three declared one-dimensional arrays *a*, *b* and *c* consist of *nb* elements. Each element is a regular C-style one-dimensional array of *blockn* double precision floating-point values. Array elements can be referenced using *index expressions*. Index expression is a ParSolGen expression of the form *a{v}[e1]..[eN]*, where *a* is the name of an N-dimensional array, *e1, ..., eN* are ParSolGen expressions called *subscripts*. The subscripts determine the referenced element indices. The expression *v* is a ParSolGen expression that specifies the version of the array element (referred to as *version reference expression*). If no version reference expression is specified, the version of the array element is assumed to be zero. For example in the above code snippet the expression *a[i]* references the i-th element of the array *a*.

The first for-loop statement describes a set of $2 * nb$ operations, each associated with the kernel *init_vec* using two *operation declaration statements*. Each operation takes values of the variables *i* and *blockn* as input and computes the value of the *i*-th element of array *a* (or *b* correspondingly). The versions of the referenced array elements are assumed to be zero. These two operation declaration statements form the *body* of the corresponding for-loop (similarly, a body of a while-loop can be considered). The expressions *i* and *blockn* are called the *input arguments* of the operation declaration statement, the expression *a[i]* is called the *output argument* of the statement. Similarly, the second for-loop statement describes a set of *nb* operations, each associated with *vec_sum* kernel. These operations compute the values of the *i*-th elements of the array *c* by the given value of the variable *blockn* and the values of the *i*-th elements of the arrays *a* and *b*.

Any ordered subset of ParSolGen statements can be illustrated as a DAG of operations and variables. "Fig. 1" illustrates the DAG representation (referred to as SSGR - Statements Subset Graph Representation) of the given example numerical algorithm description.

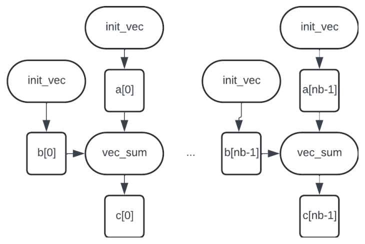

Fig. 1. SSGR example

In the figure, rectangles represent variables, ovals represent operations.

Consider another more complicated example. The below code snippet illustrates the ParSolGen description of the sparse CSR (Compressed Sparse Row) [16] matrix-vector product algorithm: $y = A * x$, where *x* is the input vector, *A* is the input CSR matrix and *y* is the output vector storing the result. Vectors *x* and *y* are represented as one-dimensional arrays of vector blocks (they store *nb* vector blocks, with each block being a one-dimensional array of *blockn* double precision floating point values). The input CSR matrix (*A*) is represented as a 2-dimensional array of matrix blocks each stored in CSR format.

```
//...

Array<int> ia[n+1];
Array<int> ja[nnz];
Array<double> a[nnz];

//...

BlockedCSRMatrix<double[][]> A = {n, n,
        nnz, nb, nb, ia, ja, a};

Array<double[blockn]> x[nb];
Array<double[blockn]> y[nb];

//...

for (int i = 0; i < nb; ++i) {
        int v = 0;
        for (int j = 0; j < nb; ++j) {
                if (!%zero(A[i][j])) {
                        csr_mv(nb,
                          blockn,
                          A[i][j],
                          x[j],
                                    y{v}[i],
                                    y{v+1}[i]);
                                v = v + 1;
                    }
            }
}
```

In the example code snippet the input matrix is initialized using standard CSR arrays: *ia*, *ja* and *a*. Each CSR matrix block can be checked for emptiness using *%zero* expression which is *true* if the CSR matrix block only stores zero values. The kernel *csr_mv* computes the following expression: $y[i] = y[i] + A[i][j] * x[i]$, where *A[i][j]* is a CSR matrix block, *y[i]* and *x[i]* are the corresponding input and output vector blocks. The use of different versions of each *y[i]* vector block enables ParSolGen compiler to store the values of these formally independent vector blocks in the same memory location. This enables overwriting the values of *y* vector reducing the potential run-time overhead introduced by the management of vector block instances.

V. PARSOLGEN ARCHITECTURE

ParSolGen consists of two parts: a classifying compiler and a set of domain-specific run-time systems. ParSolGen compiler parses the input numerical algorithm description written in ParSolGen language and analyses it for lexical and syntax errors. The it feeds the analyzed numerical algorithm description into the input of the so called classifier. The classifier determines the class to which each statement of the input numerical algorithm description belongs. As a result, the input numerical algorithm description is divided into non-intersecting, ordered subsets of statements, with each subset containing statements of the same class (referred to as classified statement subsets). For each supported statements class there is a domain-specific parallel program construction module integrated into the compiler. A parallel program part is constructed for each classified statements subset using the corresponding domain-specific parallel program construction module. Each of these program construction modules is suited only to a specific, limited class of statements and, consequently, to a restricted class of numerical algorithms (for example, sparse linear algebra or stencil based numerical algorithms). The constructed program part is linked to the corresponding domain-specific run-time system if needed (for example, in case if static code generation is not possible or there is a need for dynamic load balancing support). The described process of the parallel program construction is illustrated in "Fig. 2".

A. Numerical algorithm description classification

The ParSolGen compiler classifies the input numerical algorithm description by applying a set of integrated domain-specific classification modules (one for each supported numerical algorithms class) to each statement. Each classification module can determine if a statement belongs to the supported class. Let C_i be *i*-th classification module and $C_i(s) \in \{true, false\}$ denote the result of the application

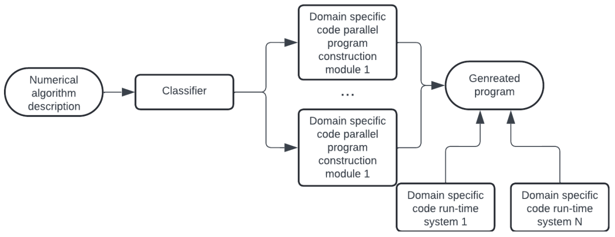
Numerical algorithm description
Classifier
Domain specific code parallel program construction module 1
...
Domain specific code parallel program construction module 1
Genreated program
Domain specific code run-time system 1
Domain specific code run-time system N

Fig. 2. The process of parallel program construction in ParSolGen

of the classification module C_i to the statement $s \in S$ where S is an ordered set of ParSolGen statements representing the input numerical algorithm description in ParSolGen language. PsrSolGen classification algorithm is listed in "Alg. 1".

Algorithm 1 An algorithm with caption

```
$C \leftarrow \{c_0, ..., c_m\}$    ▷ Ordered set of the domain specific classification modules integrated to ParSolGen
$S \leftarrow \{s_1, ..., s_n\}$    ▷ Ordered set of ParSolGen statements representing the input numerical algorithm description
$R \leftarrow \emptyset$    ▷ Statements classification
$i \leftarrow 1$    ▷ Index of the current ParSolGen statement
$k \leftarrow 1$    ▷ Index of the current classifier
while $i \leq |S|$ do
    $s_i \leftarrow next\ statement\ from\ S$
    $m \leftarrow 1$
    $r \leftarrow false$    ▷ Result of the application of the k-th classification module to the i-th statement
    while $m \leq |C| \wedge r = false$ do
        if $c_k(s_i) = true$ then
            $R \leftarrow R \cup \{\langle s_i, k \rangle\}$
            $r \leftarrow true$
        else
            if $k = |C|$ then
                $k \leftarrow 1$
            else
                $k \leftarrow k + 1$
            end if
        end if
        $m \leftarrow m + 1$
    end while
    if $r = false$ then return error    ▷ i-th statement is not classified - return error
    end if
    $i \leftarrow i + 1$
end while
```

The idea behind the classification algorithm is to find the integrated domain-specific parallel program construction module which is suitable for the current statement. This is done by looping through all the integrated classification modules until the *k*-th module returns *true*. This means that the statement belongs to the class defined by the classification module and the corresponding *k*-th parallel program construction module will be used to construct the corresponding part of the output parallel program. If a statement cannot be classified, an error is thrown and the parallel program construction is aborted. The above classification algorithm allows the accumulation of domain-specific classification and parallel program construction modules in ParSolGen which makes it practical.

Each classification module implements criteria that define a specific class of ParSolGen statements. Before considering the criteria defining the supported ParSolGen statement classes the following two definitions should be introduced.

Definition 1. A ParSolGen expression is called *trivial* if it meets the following criteria:

1) An integer constant
2) An integer variable reference
3) $e_1 + e_2$, where e_1 and e_2 are trivial ParSolGen expressions
4) $e_1 - e_2$, where e_1 and e_2 are trivial ParSolGen expressions
5) $e{*}c$, where e is a trivial ParSolGen expression and $c \in \mathbb{Z}$ is an integer constant

Definition 2. A ParSolGen index expression is called *trivial* if all its subscripts are trivial and its version reference is trivial.

For example the following expressions are trivial: $a + 1$, $a + b - 1$, $A1[i - 1][1]$, where *a*, *b* and *i* are the integer variable names, *A* is the name of a two-dimensional array. The expression $a * A[0] - 3.5$ is not trivial.

In this paper two supported ParSolGen statement classes are considered. The first supported statements class is referred as ”dense linear algebra related statements class”. A ParSolGen statement is considered as a dense linear algebra related statement if the following conditions are met:

- Operation declaration statement should meet the following criteria:
  1) The statement either has a single output argument referencing an array element or all its output arguments referencing array elements have pairwise-equal subscript expressions
  2) All arguments are trivial expressions
  3) If an argument references an array element, the array is two-dimensional
  4) At least one input argument is an index expression
  5) None of the arguments reference a sparse matrix blocks
- For-loop statement body contains only operation declaration statements that meet the above requirements

Satisfying the above conditions does not necessarily guarantee that a statement describes a part of a dense linear algebra related algorithm. In case if a statement satisfies the above conditions, a parallel program part for the statement will be constructed using the program construction module designed for dense linear algebra domain. In case of miss-classification the parallel program remains correct. However, the domain-specific optimizations and techniques employed in the process of the parallel program construction can potentially reduce the performance of the constructed parallel program part.

Consider the second supported statements class. It is referred to as ”sparse linear algebra related statements class”. The criteria defining this class are the following:

1) Operation declaration statement satisfies the following criteria:
   a) All arguments are trivial expressions
   b) At least one of the input arguments is an index expression referencing a sparse matrix block
   c) If an output argument is an index expression, it references a one-dimensional array element
2) For-loop statement body contains only operation declaration statements that meet the above requirements

The notes provided for dense linear algebra-related statements also apply to the class described above.

Consider an example ParSolGen code snippet.

```
...
int v1 = 1;
int v2 = 2;
Array<double[blockn]> x[nb];
BlockedCSRMatrix<double[][]> A = ...;
...
for (int i = 0; i < nb; ++i)
{
        // assume that the first
        // two arguemnts are input
        // and the third one is output
        some_csr_operation(
          A{1}[i][v1],
          i,
          x[i]);
}
```

Consider the for-loop statement body. It contains a single operation declaration statement associated with *”some_csr_operation”* kernel. The statement doesn’t meet the criteria for dense linear algebra related statements class: the first input argument references a CSR matrix block. However, it meets the criteria for sparse linear algebra related statements class: the first argument is an index expression that references a block of the CSR matrix *A*, all arguments are trivial and the last output argument is an index expression that references an element of the array *x* which is one-dimensional. Therefore, according to the above criteria description the considered for-loop statement also belongs to the sparse linear algebra related statements class.

The above criteria together define a ParSoGen numerical algorithm descriptions class which includes various popular and widely-used dense and sparse linear algebra algorithms such as: dense LU, LL^T, LDL^T matrix factorization algorithms, various sparse iterative solver algorithms (Conjugated Gradient, Biconjugated gradient, Generalized Minimal Residual method etc.) and various sparse matrix preconditioner algorithms (Jacobi, Successive Over-Relaxation, etc.). This makes the described classification algorithm and criteria practical.

VI. ParSolGen parallel program construction

A parallel program part is constructed for each classified statements subset. This is done by the corresponding domain-specific parallel program construction module integrated into ParSolGen compiler.

For dense and sparse linear algebra related statements two staged parallel program construction process is applied. First so called *intermediate* program is constructed. The intermediate program builds SSGR at runtime. This SSGR is then fed into the input of the corresponding domain-specific run-time system that executes it in a distributed manner. The process is illustrated in "Fig. 3"

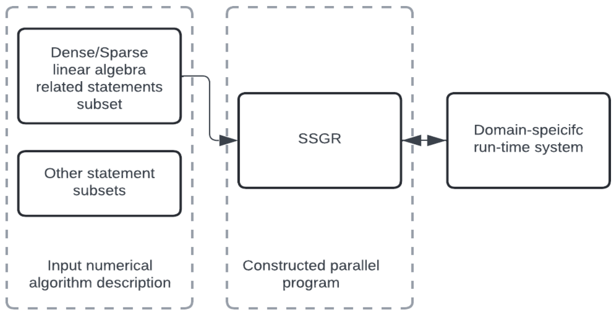
Dense/Sparse linear algebra related statements subset
Other statement subsets
Input numerical algorithm description
SSGR
Constructed parallel program
Domain-speicifc run-time system

Fig. 3. Two-staged parallel program construction process

Two domain-specific runtime systems are considered. These run-time systems are able to execute the dense and sparse linear algebra related SSGRs correspondingly. The architecture of both domain specific runtime systems is shown in "Fig. 4".

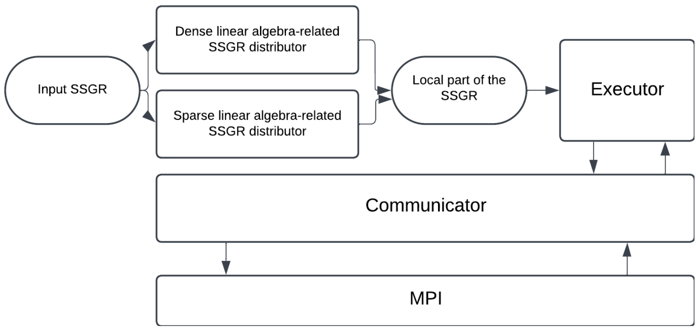
Input SSGR
Dense linear algebra-related SSGR distributor
Sparse linear algebra-related SSGR distributor
Local part of the SSGR
Executor
Communicator
MPI

Fig. 4. Domain-specific run-time systems architecture

The process of the SSGR execution consists of two steps. First, the input SSGR is distributed among the computing nodes of the target parallel computer which means that:

- For each operation in the SSGR, the computing node on which to execute the operation is determined
- For each variable in the SSGR, the computing node where the variable's value will be stored is determined

This is done by the corresponding domain-specific distributor module. Then the part of the SSGR which is mapped to a certain computing node is executed by the corresponding domain-specific executor module. When an operation in the SSGR is executed by the executor the following steps are performed for each operation output variable:

- All consumers (operations that take the variable as input) of the variable are found in the SSGR
- If a consumer of the variable is mapped to a different computing node, the variable value is sent asynchronously to the target node by the communicator module
- The corresponding domain-specific executor module is notified that the variable value is computed and available

The communicator module runs a separate thread, continuously checking in an infinite loop for any variable values ready to be received. When the communicator module receives the value of a variable, the variable is marked as computed, all domain-specific executors are notified.

A. Sparse linear algebra related SSGR distribution

The implemented domain-specific algorithm for distributing a sparse linear algebra related SSGR can be explained using an illustrative example. Let A be a sparse matrix represented as a 2-dimensional ParSolGen array of CSR matrix blocks. Consider the example SSGR shown in "Fig. 5".

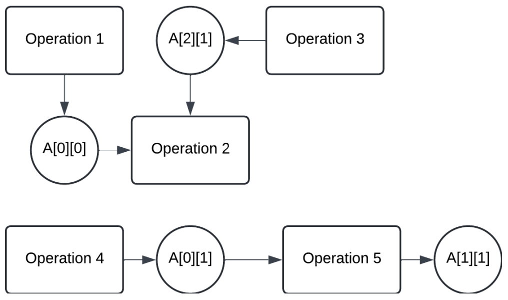

Fig. 5. Example sparse linear algebra related SSGR

The notation $A[i][j]$ references an element of the matrix A with the indices i and j. Squares denote operations, circles - the blocks of the matrix A.

First, all variables are removed from the input SSGR in a way that each operation computing a variable value is directly connected to all consumers of the variable. Then, for each arc in the resulting directed graph, an arc in the opposite direction is added as shown in "Fig. 6". The resulting graph is referred to as the *operations graph*.

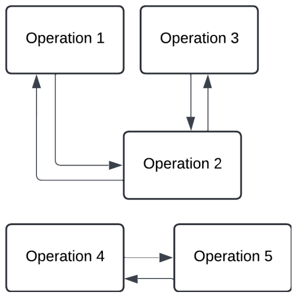

Fig. 6. Operations graph example

After the operations graph is built, it is partitioned into k non-intersecting sub-graphs, where k is the number of the computing nodes in the target parallel computer. This is done using the implementation of a k-way graph partitioning algorithm from METIS [18] library. The example distribution of the above operations graph (with $k = 2$) is shown in "Fig. 7".

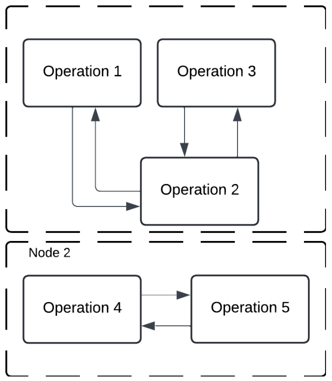

Fig. 7. Operations graph distribution example

The final step of the distribution algorithm is to distribute the variables of the input SSGR among the nodes of the target parallel computer. A simple rule is applied: each variable is assigned to the same computing node as the operation that computes it (the variable's producer). The example distribution of the considered SSGR is shown in "Fig. 8".

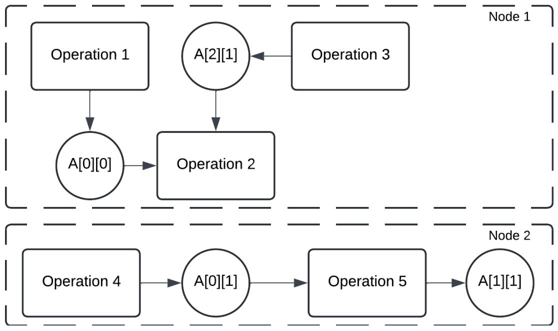

Fig. 8. The distribution of the sparse linear algebra related SSGR

B. Dense linear algebra related SSGR distribution

Consider the implemented algorithm for distributing a dense linear algebra related SSGR. Similar to the previous section, the algorithm is explained using an example SSGR. Let A be a 2-dimensional dense matrix represented as a 2-dimensional ParSolGen array of dense matrix blocks. Consider the example SSGR shown in "Fig. 9".

First, the widely used block-cyclic approach [19] is applied to distribute all dense matrices referenced by in the input SSGR. The example block-cyclic distribution of the matrix A is shown in "Fig. 10".

The notation *A[i][j]* denotes an element of the matrix A with the indices i and j. Different colors represent the different computing nodes to which the matrix elements are mapped. After all the matrices are distributed, each operation of the

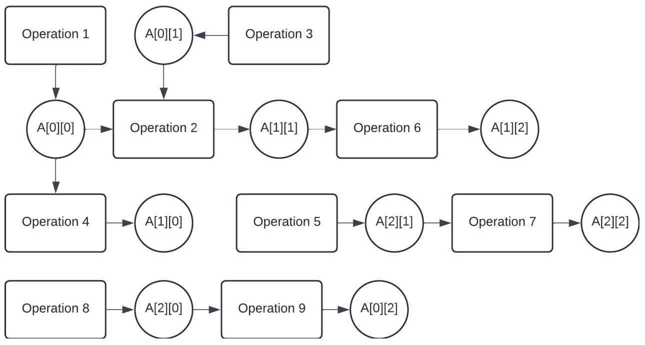

Fig. 9. Example dense linear algebra-related SSGR

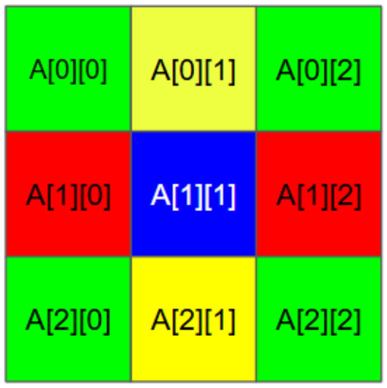

Fig. 10. The example of the block-cyclic matrix distribution

input SSGR is assigned to the same computing node as the matrix elements referenced by its output arguments. The distribution of the considered example SSGR is illustrated in "Fig. 11".

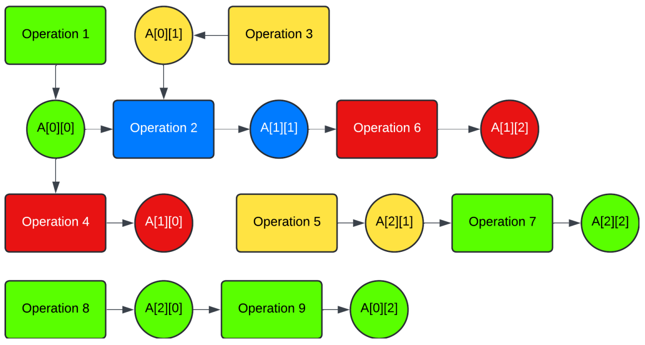

Fig. 11. The distribution of the dense linear algebra related SSGR

C. The SSGR execution algorithm

After the input SSGR is distributed, its local part (a sub-graph of the SSGR which is assigned to the specific computing node) is executed. This is done using the following algorithm:

- Each operation within the local part of the input SSGR which either has no input variables or has all input variables computed is submitted to multi-threaded tasks queue for execution

- After an operation is executed, all its output variables are marked as computed
- The process repeats until all operations from the local part of the input SSGR are executed

The above algorithm is used to execute the local parts of both dense and sparse linear algebra related SSGRs. Despite its simplicity, it enabled achieving performance in the constructed parallel programs comparable to that of popular library implementations of widely-used linear algebra algorithms. This makes the simple algorithm under consideration practical.

VII. Performance evaluation

The performance of the parallel programs constructed by ParSolGen is evaluated on four test applications: distributed dense matrix LL^T factorization, distributed sparse CSR matrix-vector product, distributed sparse CG (Conjugate Gradient) iterative solver and distributed sparse BiCG (Biconjugate Gradient) iterative solver. The performance of the parallel programs constructed by ParSolGen was compared to that of the popular libraries implementing the same algorithms.

A. Distributed dense matrix LL^T factorization

A parallel program was constructed by ParSolGen from the description of the algorithm of LL^T factorization of a dense matrix. The performance of the parallel program constructed by ParSolGen was compared to that of ScaLAPACK (v2.2.0) implementation of LL^T factorization. Execution times of both implementations were compared. OpenBLAS [20] library (version 0.3.15) [15] implementation of BLAS and LAPACK subroutines was used for both tests.

Two square dense double-precision matrices of sizes 32768 and 65536 were used as input data. For each of the matrices a number of experiments were conducted using different matrix block sizes ranging from 256 to 2048. “Fig. 12” and “Fig. 13” shows execution times comparison of the ScaLAPACK and the ParSolGen implementations for the input matrix of sizes 32768 (left) and 65536 (right).

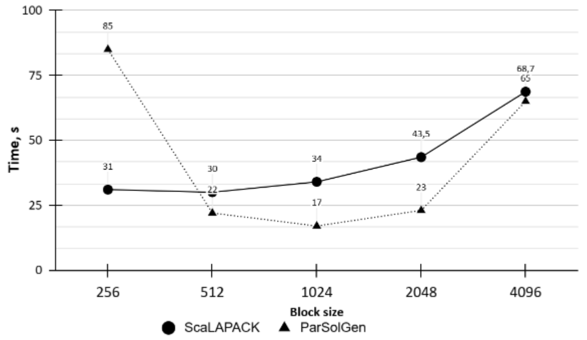

Fig. 12. Performance evaluation result for a square matrix of 32768 elements.

Testing was conducted on a two-dimensional grid of 2×2 computing nodes. Each node consists of 32 cores and 16 GB of memory. All 32 cores of each CPU were used in all tests.

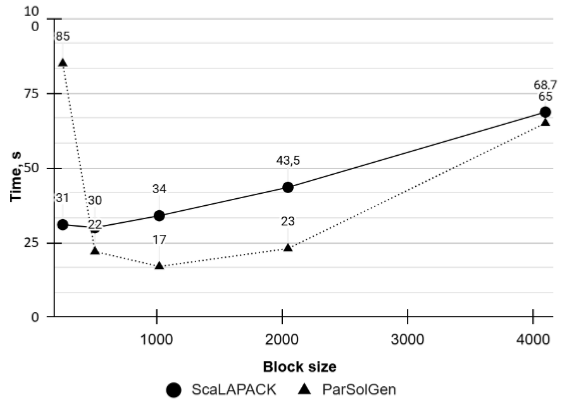

Fig. 13. Performance evaluation result for a square matrix of 65536 elements.

B. Parallel sparse matrix-vector product

A parallel program was constructed by ParSolGen from the description of CSR matrix-vector product algorithm. The performance of the parallel program constructed by ParSolGen was compared to that of Intel MKL (v2021.4.0) on single node (Intel 12th Gen Intel(R) Core(TM) i7-12700H, 64GB RAM) running Ubuntu 22.10. The matrix ”VLSI/stokes” CSR matrix from the Sparse Matrix Collection [17] was used as input data. Both implementations computed 1000 sparse matrix vector products and the execution time was measured. The results are shown in “Fig. 14”.

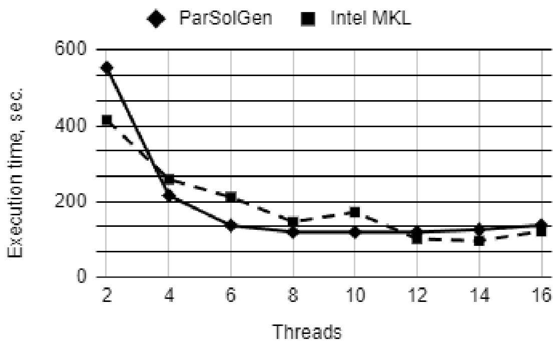

Fig. 14. ParSolGen vs Intel MKL performance comparison

Here the performance of the parallel program constructed by ParSolGen is comparable to that of Intel MKL implementation.

C. Sparse BiCG solver

A parallel program was constructed by ParSolGen from the description of CSR matrix BiCG (Bi-Conjugated Gradient) iterative solver algorithm. The performance of the parallel program constructed by ParSolGen was compared to that of PETSc (v3.22.1) on a single Google Cloud C2 node (30 Intel Cascade Lake virtual cores and 240 GB RAM) running Amazon Linux. Two tests were conducted on two matrices from Sparse Matrix Collection [17]: ”VLSI/stokes” and ”VLSI/nv2”. Both implementations executed 100 iterations of the BiCG algorithm on ”VLSI/stokes” matrix and 1000 iterations on ”VLSI/nv2”. The results are shown in “Fig. 15” and “Fig. 16”.

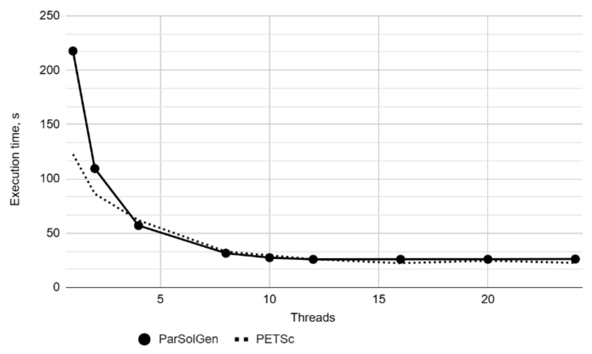

Fig. 15. ParSolGen vs PETSc performance comparison (matrix ”VLSI/stokes”)

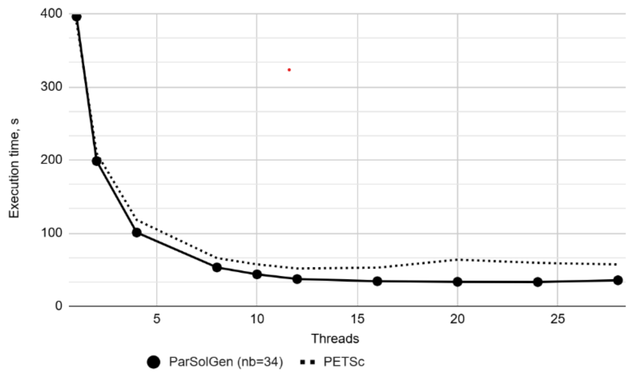

Fig. 16. ParSolGen vs PETSc performance comparison (matrix ”VLSI/nv2”)

Here the performance of the parallel program constructed by ParSolGen is comparable to that of PETSc implementation of BiCG iterative solver algorithm.

VIII. Conclusion

Automatic parallel program construction requires different construction system algorithms and techniques for different subject domains. ParSolGen is an automated parallel program construction tool that is designed for the accumulation of various domain-specific automatic parallel program construction algorithms. This ability was demonstrated by integrating specialized parallel program construction algorithms for dense and sparse linear algebra domains into ParSolGen. The performance of the parallel programs automatically constructed by ParSolGen from the description of sparse matrix-vector product, dense LL^T matrix factorization, sparse CG (Conhugated Gradient) and sparse BiCG (Biconjugate gradient) algorithms is comparable to that of widely used libraries. This makes ParSolGen a practically useful tool for the automatic construction of highly efficient distributed parallel programs for sparse iterative solvers and dense linear algebra subject domains. Other numerical algorithm classes can also be supported by ParSolGen by integrating the corresponding domain specific parallel program construction algorithms.

ParSolGen is available for download as a binary package at [2], [3].

References

[1] Bosilca, George, et al. ”Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA.” 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. IEEE, 2011.
[2] https://github.com/parsolgen/ParSolGenHome
[3] https://parsolgen.github.io
[4] Bauer, Michael, et al. ”Legion: Expressing locality and independence with logical regions.” SC’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE, 2012.
[5] Kale, Laxmikant V., and Sanjeev Krishnan. ”Charm++ a portable concurrent object oriented system based on c++.” Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications. 1993.
[6] Kale, Laxmikant V., and Sanjeev Krishnan. ”Charm++.” (1996).
[7] AsimYarkhan.2012. DynamicTaskExecutiononSharedandDistributed-Memory Architectures. December (2012). http://trace.tennessee.edu/utk
[8] Augonnet, Cédric, Samuel Thibault, and Raymond Namyst. StarPU: a runtime system for scheduling tasks over accelerator-based multicore machines. Diss. INRIA, 2010.
[9] Ragan-Kelley, Jonathan, et al. ”Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines.” Acm Sigplan Notices 48.6 (2013): 519-530.
[10] Hoque, Reazul, et al. ”Dynamic task discovery in parsec: A data-flow task-based runtime.” Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems. 2017.
[11] Bosilca, George, et al. ”DAGuE: A generic distributed DAG engine for high performance computing.” Parallel Computing 38.1-2 (2012): 37-51.
[12] Slaughter, Elliott. Regent: A high-productivity programming language for implicit parallelism with logical regions. Diss. Stanford University, 2017.
[13] Slaughter, Elliott, et al. ”Regent: a high-productivity programming language for HPC with logical regions.” Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2015.
[14] Belyaev, Nikolay, and Vladislav Perepelkin. ”High-efficiency specialized support for dense linear algebra arithmetic in LuNA system.” Parallel Computing Technologies: 16th International Conference, PaCT 2021, Kaliningrad, Russia, September 13–18, 2021, Proceedings 16. Springer International Publishing, 2021.
[15] Belyaev, Nikolay, and Vladislav Perepelkin. ”High-efficiency specialized support for dense linear algebra arithmetic in LuNA system.” Parallel Computing Technologies: 16th International Conference, PaCT 2021, Kaliningrad, Russia, September 13–18, 2021, Proceedings 16. Springer International Publishing, 2021.
[16] https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-dpcpp/2024-1/sparse-storage-formats.html#SPARSE-STORAGE-FORMAT-CSR
[17] Davis, Timothy A., and Yifan Hu. ”The University of Florida sparse matrix collection.” ACM Transactions on Mathematical Software (TOMS) 38.1 (2011): 1-25.
[18] Karypis, George, and Vipin Kumar. ”METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices.” (1997).
[19] Prylli, Loïc, and Bernard Tourancheau. ”Efficient block cyclic data redistribution.” Euro-Par’96 Parallel Processing: Second International Euro-Par Conference Lyon, France, August 26–29 1996 Proceedings, Volume I 2. Springer Berlin Heidelberg, 1996.
[20] https://www.openblas.net/