Solving Linear Diophantine Systems on Parallel Architectures

Dmitry Zaitsev, *Senior Member, IEEE*, Stanimire Tomov, and Jack Dongarra, *Fellow, IEEE*

Abstract—Solving linear Diophantine systems of equations is applied in discrete-event systems, model checking, formal languages and automata, logic programming, cryptography, networking, signal processing, and chemistry. For modeling discrete systems with Petri nets, a solution in non-negative integer numbers is required, which represents an intractable problem. For this reason, solving such kinds of tasks with significant speedup is highly appreciated. In this paper we design a new solver of linear Diophantine systems based on the parallel-sequential composition of the system clans. The solver is studied and implemented to run on parallel architectures using a two-level parallelization concept based on MPI and OpenMP. A decomposable system is usually represented by a sparse matrix; a minimal clan size of the decomposition restricts the granulation of the technique. MPI is applied for solving systems for clans using a parallel-sequential composition on distributed-memory computing nodes, while OpenMP is applied in solving a single indecomposable system on a single node using multiple cores. A dynamic task-dispatching subsystem is developed for distributing systems on nodes in the process of compositional solution. Computational speedups are obtained on a series of test examples, e.g., illustrating that the best value constitutes up to 45 times speedup obtained on 5 nodes with 20 cores each.

Index Terms—MPI, OpenMP, linear diophantine system, Petri net, clan, speed-up

1 Introduction

Solving linear Diophantine systems is applied in a wide range of disciplines, such as in discrete-event systems [1], model checking [2], formal languages and automata [3], logic programming [4], cryptography [5], networking and signal processing [6], and chemistry [7].

Petri nets [1] represent a discrete-event system widely applied for verification of communication protocols; evaluating network performance; manufacture control and business processes managements; solving tasks in chemistry and biology; and modeling concurrent computations. There are two basic approaches to check models represented by Petri nets. These are based on: (1) the model state space (called a reachability graph), and (2) solving linear Diophantine algebraic equations and inequalities. Other auxiliary techniques, such as reduction and decomposition, are also applied. In most cases, solving a Diophantine system, either homogeneous or heterogeneous, requires results in non-negative integer numbers, which represents an intractable problem. Further, it is rather difficult to predict the number of basis solutions and the system solving time based on its size [8].

- *D. Zaitsev was with the Department of Computer Engineering and Innovative Technology, International Humanitarian University, Fontanskaya Doroga 33, Odessa 65009, Ukraine. E-mail: daze@acm.org.*
- *S. Tomov and J. Dongarra are with the Innovative Computing Laboratory, The University of Tennessee, Knoxville 37996, TN. E-mail: {tomov, dongarra}@icl.utk.edu.*

Manuscript received 5 Mar. 2018; revised 13 Sept. 2018; accepted 20 Sept. 2018. Date of publication 5 Oct. 2018; date of current version 10 Apr. 2019. (Corresponding author: Dmitry Zaitsev.)
Recommended for acceptance by P. Balaji.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2873354

Solving a homogeneous Diophantine system in non-negative integers is used for such powerful Petri net analysis techniques as linear invariants of places and transitions, siphons, and traps. Note that solving heterogeneous systems and systems of inequalities is reduced to solving homogeneous systems [9]. For solving a homogeneous system, an algorithm was proposed by Toudic [10] and further refined by Colom and Silva [11]. The algorithm was implemented in manifold software systems for Petri nets analysis, including module Adriana [1], developed by the first author as a plugin for system Tina [12].

In [13], a technique has been introduced to speed up solving a linear system involving decomposition into subsets of equations called clans. This technique can be considered supplementary to the traditional methods for parallel solving of dense [14] and sparse [15] systems. Decomposability of a matrix into clans is its intrinsic property, based on the sign of its elements. As a result of the decomposition, a set of clans is obtained. The minimal clan size limits the granularity of the technique since after the decomposition we can only unite some clans to obtain bigger ones. Possible variance in clan size introduces some initial imbalance. A speedup is obtained as a consequence of sets of systems, having lesser dimensions, being solved instead of solving the source system directly.

In this paper, we describe a recent implementation of the decomposition into clans [13] on modern parallel architectures using multiple nodes with MPI communications between them [16], [17], [18], and OpenMP [19], [20] to extract additional parallelism within the nodes' multiple cores. The software developed is called ParAd (Parallel Adriana). For the parallel implementation of ParAd, we transform the sequential composition of clans specified in [13] into a parallel-sequential composition to employ

multiple computing nodes. To use OpenMP, we transform the Toudic algorithm to provide maximal possible independence of passages for basic loops. To compensate for the initial imbalance of the decomposition into clans, we develop and verify protocols of dynamic scheduling of tasks for MPI nodes. Using multi-core architectures provides up to $15\times$ speedup on 20 cores versus one core, while the composition yields an additional acceleration of 2–3$\times$ on 5 nodes.

2 Solving Systems via Composition of Their Clans

The technique of solving systems via composition of their clans (functional subnets) has been developed first for Petri nets [21], [22] and then generalized on an algebraic structure of rings with a sign [13].

Let us consider a homogeneous linear system

$$A\vec{x} = 0,$$

over a ring with a sign, and suppose that there is an algorithm to find its general solution and to represent it in the following form:

$$\vec{x} = G\vec{y},$$

where G is a matrix of basis solutions and $\vec{y}$ is a vector of independent free variables.

A *nearness relation* on the set of the system equations is defined in the following way: two equations are near if they contain at least one variable with a nonzero coefficient of the same sign. A *clan relation* is defined as a transitive closure of the nearness relation, and a subset obtained as a result of equation partitioning is called *a clan*. After the decomposition into k clans, a system matrix is represented as a union of a block-column matrix and a block-diagonal matrix [13]

$$A = \begin{bmatrix} \breve{A}_1 & \hat{A}_1 & 0 & ... & 0 \\ \breve{A}_2 & 0 & \hat{A}_2 & ... & 0 \\ ... & ... & ... & ... & ... \\ \breve{A}_k & 0 & 0 & ... & \hat{A}_k \end{bmatrix},$$

where an ith block-row matrix represents a clan A_i that is specified by a pair of nonzero matrices $A_i = (\breve{A}_i, \hat{A}_i)$. The block-column matrix $\breve{A} = (\breve{A}_1, \ldots, \breve{A}_k)^T$ specifies connection of clans and the corresponding variables are called *contact variables*; the block-diagonal matrix corresponds to the *internal variables* of clans defined by $\hat{A}_i$. It has been proven [13] that a contact variable belongs to exactly two clans, wherein one clan contains it with a positive sign and the other clan contains it with a negative sign. As a consequence, a decomposable matrix is a sparse one. The complexity of the decomposition algorithm is linear in the number of nonzero elements of matrix A. Thus, for a contact variable, represented by a column of matrix $\breve{A}$, nonzero elements are situated in exactly two matrices, e.g., $\breve{A}_i$ and $\breve{A}_j$ entering one of them with the positive sign and the other with the negative sign. Based on this property, the idea for writing the composition system consists in equating the variable values obtained using bases of the two mentioned clans.

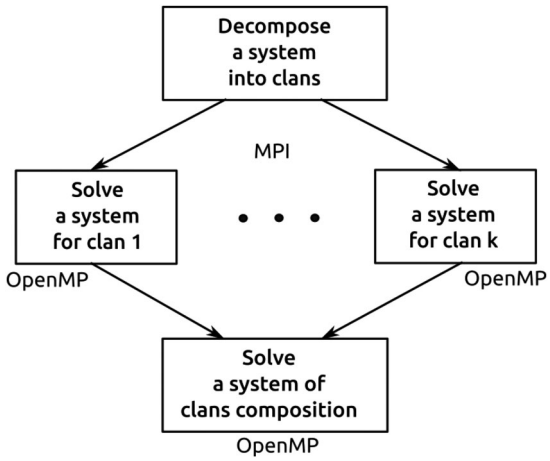

Fig. 1. Solving a system with simultaneous composition of its clans.

The compositional algorithm (Fig. 1) of solving a linear system [13] consists of the following steps:

- decompose a system into its clans;
- obtain a general solution for each clan $1 \le j \le k$, where k is the number of clans

$$\vec{x}^j = G^j\vec{y}^j;$$

- solve the composition system

$$F\vec{y} = 0,$$

  where matrix F is obtained for variables connecting a pair of clans in such a way that their values, computed according to each clan basis solution, coincide; its solution is represented as $\vec{y} = R\vec{z}$;
- recover the source system solution as

$$\vec{x} = H\vec{z},$$

  where $H = GR$ and G is a joint matrix of solutions for clans.

As demonstrated in [13], the compositional algorithm enables a computational speedup because it results in solving a set of systems of lesser dimension that can also be easily implemented and computed in parallel.

Note that the clan size limits the granularity of this technique because the decomposition algorithm produces a set of the minimal clans, which represents a basis for the clans structure regarding the operation of union [13]. Obtained clans of varying sizes introduce some imbalance, which should be mended during the compositional solution.

When the composition system is rather big compared to clans, it was advised [13] to implement the composition in a sequential way; the corresponding process has been represented as *a collapse* of the decomposition graph. The most fine granulation is obtained for a pairwise (edge) collapse, where at each step a system is solved, corresponding to the composition of a pair of connected clans (Fig. 2). In the decomposition graph, a vertex corresponds to a clan and an edge corresponds to the contact variables entering both clans; the edge weight equals the number of common contact variables. At each step of collapse, an edge is contracted

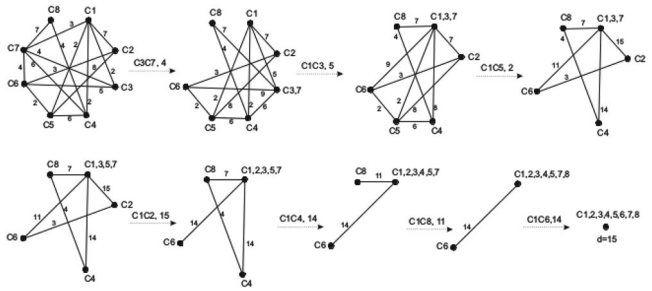

Fig. 2. Solving a composition system sequentially (edge collapse of a weighted graph).

that corresponds to solving the composition system for the pair of clans connected by the edge.

To gain an additional speedup from a fast implementation of the approach on parallel architectures, we here offer a parallel-sequential process of the decomposition graph collapse. It is rather difficult to formalize the task completely in terms of graph theory because systems started in parallel are not solved at the same time. This is especially significant when solving a system in non-negative integer numbers due to somewhat unpredictable solve time. An example of such a parallel-sequential collapse is shown in Fig. 3; its width is 20, and the maximal number of workers is 4. Contracting 4 edges on the first step means solving 4 systems in parallel by 4 workers; on the second step, 2 systems are solved; on the third step, 1 system is solved. However, organizing the process by steps seems inefficient because of the various time it takes for solving systems even of the same size.

Therefore, because of the various clan sizes and unpredictable system solve times, we implement a dynamic scheduling approach based on a greedy strategy of an edge choice. A greedy strategy works rather well for the sequential collapse [13]. When starting jobs for solving systems on edges in parallel, we should use an *independent set* of the graph edges [23]. The dynamic scheduling uses two subgraphs: a current working subgraph D and a subgraph containing the set of contracting edges P. If there are free MPI workers, an independent edge (together with edges of P) is chosen, removed from D, and added to P, and the corresponding job is started. When there are no independent edges in a graph $D \cup P$ and both graphs are not empty, we should wait until an ongoing job finishes. When a job

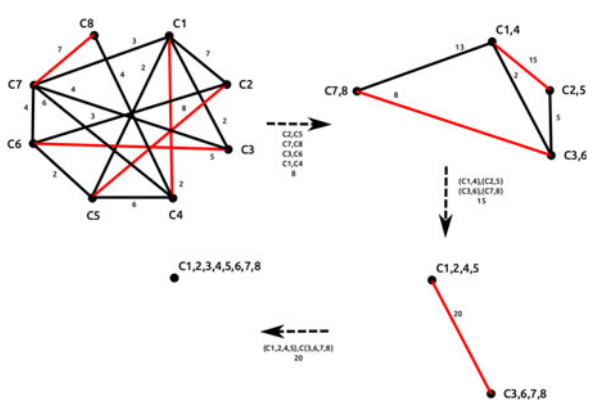

Fig. 3. Solving a composition system parallel-sequentially.

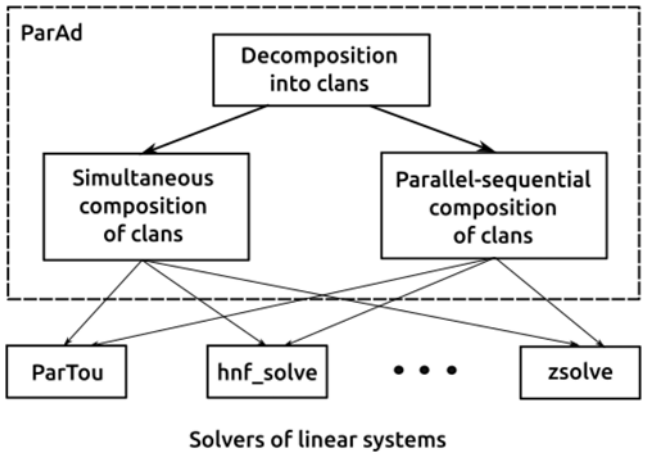

Fig. 4. The general scheme of ParAd's layout.

finishes, the corresponding edge is removed from P, returned to D, and contracted according to the rules specified in [13]. When an edge is contracted, its two vertices are replaced by a single vertex, and sets of their incident edges merge; contraction of a triangle leads to obtaining edges with weights equal to a sum of weights for edges adjacent to the contracted edge.

3 Overall Organization of ParAd

We implement the technique of the system (matrix) decomposition into its clans, and, further, solve the corresponding linear homogeneous system of equations through composition of clans [13] in the domain of integer numbers (Diophantine systems) independently from employed solvers of systems. This enables us to use different solvers, compare them, and solve specific tasks such as obtaining non-negative solutions valuable for analysis of Petri net. The general scheme of the ParAd layout is shown in Fig. 4.

ParAd consists of the following basic subsystems: decomposition into clans, simultaneous composition, parallel-sequential composition, and a set of solvers. We develop a specialized solver, ParTou, for obtaining non-negative solutions on multi-core architectures studied in the next section, and use a flexible solver 4ti2 [24] for the entire integer domain and its given subdomains. Other solvers can be integrated into ParAd as well, using the specified interface.

Historically, to manage solving big systems with Adriana, manifold intermediate data for clans have been stored in temporary files. We preserve this scheme in ParAd because using files allows more flexibility to attach external solvers. Note that, should optimization for a definite type of system and a set of solvers be required, the software can be easily modified for using data structures stored in RAM. Additionally, it can be implemented as a library instead of a standalone compositional solver.

4 Parallel Implementation of Toudic Algorithm Using OpenMP

The Toudic algorithm [10] finds non-negative integer solutions of homogeneous linear Diophantine systems. Originally, it was offered for finding place and transition invariants of a Petri net, which are given by the solutions of the equations

$$\vec{x}C = 0,$$

and

$$C\vec{y} = 0,$$

respectively, where C represents the incidence matrix of a Petri net [25]. Invariants play a central part in investigating structural properties of Petri nets, e.g., properties that do not depend on the initial marking. We use an improved variant of the algorithm presented by Colom and Silva [11].

Traditionally, the algorithm description is provided for finding the place invariants, and we use this specification further in the paper. Note that to find the transition invariants, a transposed matrix C is used with the same algorithm. The basic idea of the algorithm is rather simple. At each passage it cleans a column of matrix C, adding to it new rows until the obtained matrix equals zero. The same transformations on rows are applied to a unit matrix, which finally contains basis solutions. A simplified specification of the Toudic algorithm follows:

- Step 0. Form a united matrix (C, X), where $X = E$; assign $j = 0$.
- Step 1. If column j of matrix C contains coefficients of the same sign, stop; there is no solution with all the positive components.
- Step 2. Create (C', X') in the following way:
  - Step 2.1. Copy rows i where $c_{i,j} = 0$ of matrix (C, X) into matrix (C', X').
  - Step 2.2. For each pair of rows (i, i'), $c_{i,j} < 0$, $c_{i',j} > 0$ of matrix (C, X), create a new row $c'_{i'',k} = -c_{i,j}c_{i'',k} + c_{i',j}c_{i'',k}$, $x'_{i'',l} = -c_{i,j}x_{i'',l} + c_{i',j}x_{i'',l}$ of matrix (C', X').
- Step 3. Assign $(C, X) := (C', X')$, $j := j + 1$.
- Step 4. If $C \neq 0$ goto Step 1, otherwise X is a sought matrix of basis solutions.

If at each passage column j contains both positive and negative elements, the algorithm finds basis solutions; otherwise, it stops in a middle. If we are interested in basis anyway, it requires some minor amendments [11]. At Step 1, for nonzero elements of column j of matrix (C, X), the corresponding rows of matrix (C, X) are cleaned. And we proceed with the next $j := j + 1$ column, in case j is not the last column.

Some optimization improves the performance of the algorithm [11]. The coefficients $c_{i',j}$ and $c_{i,j}$ used at Step 2.2 are reduced by their greatest common divisor when obtaining the new row of matrix (C', X'). At a passage of the algorithm, instead of the next column, a column that produces the smallest number of new rows is chosen—namely, a column j with minimal value of $n \cdot p$, where n is the number of negative and p is the number of positive elements $c_{i,j}$ in the column. Finally, the obtained intermediate basis solutions X', before assigning at Step 3, are filtered with regard to the non-negative integer lattice; only maximal solutions left.

The specified algorithm, including all the mentioned amendments, was implemented in C in 2005 in the Adriana software package [1] and delivered as a plugin for system Tina [12]. Its performance is on par with the best known tools for Petri nets analysis. Regardless of this success, analysis of real-life systems in a feasible time—for instance when modeling sophisticated networking protocols and airplane control systems—requires performance improvement in a dozen and more times.

To improve performance, we choose to use OpenMP [19], [20] for the parallel implementation of the Toudic algorithm. We prefer OpenMP rather than MPI [16] because elements on a passage can be processed simply, where there would be heavy load of MPI communications of data between nodes if implemented on distributed-memory architectures. In the next section we consider decomposition of a system into clans for use on distributed-memory architectures with MPI.

First attempts to directly supply Adriana code written in C with OpenMP directives actually led to slowdown in some cases because passages of basic loops were too connected. The algorithm has therefore been completely reorganized to remove sequential dependencies as much as possible. An indicator has been added to the matrix row to specify the row presence and the number of greater elements within the lattice to filter solutions. Instead of sequiential transformations of rows, indicators are changed in parallel and then all the rows are reorganized at once in a separate loop. The row indicators allowed the copying and combining of rows, which constitutes the basic action at a step, in parallel. As a result, some minor dependencies remain for processing columns with elements of the same sign, finding the best column, and filtering solutions expressed with "#pragma omp critical" and "#pragma omp atomic." The reduction technique has been employed for summation and conjunction, for instance with "#pragma omp simd reduction(+:np,nm)".

We use the "#pragma omp parallel for" directive of the C preprocessor for the outer loops on the matrix rows for employing multiple cores for independent passages of a loop. For processing a row, we employ single instruction, multiple data (SIMD) facilities of a core with the "#pragma omp simd" directive. The following fragment of a program from ParAd illustrates the implementation of the parallel copy of the matrix rows represented in Step 2.1 of the algorithm:

```
#pragma omp parallel for private(i,k)
for( i = 0; i < nIz; i++ ){
  int *C1b = lc( C1, i+1, nc );
  int *Cbz = lc( C, Iz[i], nc );
  #pragma omp simd
  #pragma unroll
  for( k = 0; k < nc; k++ )
    C1b[k] = Cbz[k];
  int *X1b = lx( X1, i + 1, nx );
  int *Xbz = lx( X, Iz[i], nx );
  #pragma omp simd
  #pragma unroll
  for( k = 0; k < nx; k++ )
    X1b[k] = Xbz[k];
}
```

Here, matrix C' is denoted as C1 and matrix X' is denoted as X1, vector Iz contains indexes of rows (for a chosen column) the elements of which equal zero, and macros lc and lx are used to find the pointer to the first element of the specified row.

Removing zero and non-maximal rows from a matrix is rather hard work. To avoid doing this, we use the row

indicator whose value equals zero in case the row is actually present and should be processed. The useless rows are removed in an implicit way: they are ignored when creating rows of a new matrix (C', X') at Step 2.

Parallel implementation of Step 2.2 can use the "collapse (2)" directive to merge two nested loops on i and i', though this actually decreases performance. Indeed, for rather big source data, there is no available number of cores for parallel implementation of nested loops.

An arbitrary matrix contains non-zero elements located randomly that does not produce much disbalance though some individual tasks can have matrices of specific forms. Dynamical load balancing with a chunk of a dozen produces the best performance improvement of some ten percent on average. It is achieved adding "schedule (dynamic, 10)" to the corresponding "#pragma omp parallel for" directives. The obtained speedup of ParTou compared to Adriana is discussed in Section VI and illustrated with tables and diagrams. The maximal speedup achieved on a 20-core node is up to 15×.

5 Parallel-Sequential Implementation of Clan Composition on MPI

Decomposition of a Petri net into its functional subnets (clans) [21] has been applied to obtain speedup of computations when analyzing models represented by Petri nets—in particular, for verification of networking protocols [22]. In [13] the technique is presented in an abstract form applicable to a linear system over a ring with a sign. In this section we use the decomposition of a system into its clans and either simultaneous or parallel-sequential composition of clans to gain computational speedup on modern distributed-memory architectures using MPI [16].

A peculiarity of solving a Diophantine system in non-negative numbers is the unpredictability of system solving time and the number of basis solutions, which can be rather big for small systems [8]. This fact hampers application of static analysis and preparatory workload balancing. Thus, a dynamic scheduling and workload balancing have been chosen. The master represents a scheduler of jobs and also implements minor transformations of the obtained basis solutions by multiplying matrices. A worker solves a given system. This approach minimizes the transmitted data: a system is sent from the master to a worker and a basis solution is sent from a worker to the master.

Investigations using the MPI Parallel Environment (MPE) profiler reveal the fact that information exchange does not constitute a bottleneck, even using a classical MPI_Send/MPI_Recv pair for data transmission. The master is rather available to do scheduling on demand when a worker becomes free; and on the other side, the master is not too idle multiplying obtained matrices. With multicore implementation of GraphBLAS which is used for sparse integer matrix multiplication, there will be more opportunities to load cores of the master node. Compared to the time required to solve a linear Diophantine system in non-negative numbers, standard MPI communication does not represent a bottleneck.

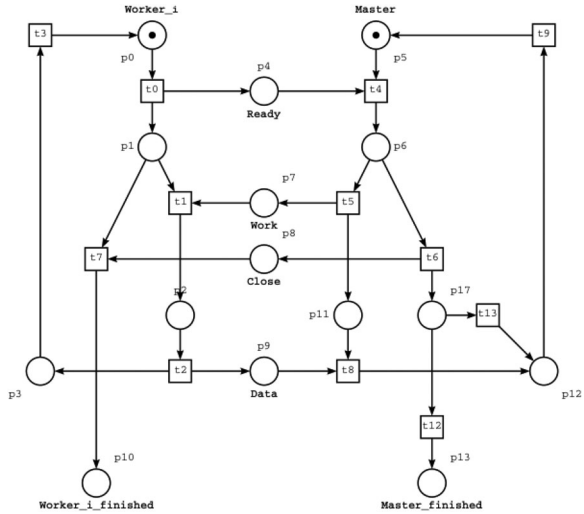

Fig. 5. Basic protocol of communicating master with workers (a Petri net).

5.1 Organization of Workers for Solving Systems

According to the compositional technique [13], a system must be solved for each of the clans, and then a composition system must be solved. When the composition system is particularly big, a collapse of the decomposition graph is implemented, and a set of systems is solved. The most fine granulation is obtained via a pairwise collapse, when a system is solved for each pair of connected clans. The most frequently required job, though—and the most complex from computational point of view—is solving a system. That is why we organize workers for solving systems only. Decomposition and other auxiliary jobs are implemented by master. We note that decomposition complexity is linear in the size of the system, and besides decomposition, matrix multiplication is required, the time complexity of which is square in the number of nonzero elements of a sparse matrix.

A basic protocol, represented in Fig. 5, has been developed for communication with workers, which suits both solving systems for clans and collapse (pairwise) of the decomposition graph. Each message is started with a single character that specifies its type; the rest of the message content depends on its type. The protocol is represented with a Petri net. A place p_6 represents a branching—either to proceed with sending jobs by t_5 or finish a worker by sending a closing message by t_6.

A worker sends an 'R' ("Ready") message, consisting of a single character, and waits tor the master reply. There are two possible variants of the master reply: 'W' ("Work"), followed by a job for the worker and 'C' ("Close") to close the worker as there is no more jobs for this worker. A job's content is organized as follows: the system number (an integer) denoted as z; the system length (an integer) denoted as lenm; the system (its sparse matrix) of lenm bytes stored in bufm. Then, solving system with a *Solver* is called (see below), and when it is finished, the worker composes a reply to the master containing the system solution. A reply is organized as follows: 'D' ("Data") an indicator that data follows; the system number (an integer) denoted as z; the basis length (an integer) denoted as lenr; the basis (its sparse matrix) of lenr bytes stored in bufr. The following simplified listing specifies the worker code:

```
while(1){
  c = 'R';
  MPI_Send(&c,1,MPI_CHAR,0,51,
                MPI_COMM_WORLD);
  MPI_Recv(&c1,1,MPI_CHAR,0,52,
                MPI_COMM_WORLD,&status);
  if(c1!='W') break;
  MPI_Recv(&z,1,MPI_INT,0,53,
                MPI_COMM_WORLD,&status);
  MPI_Recv(&lenm,1,MPI_INT,0,54,
                MPI_COMM_WORLD,&status);
  bufm = malloc(lenm);
  MPI_Recv(bufm,lenm,MPI_CHAR,0,55,
                MPI_COMM_WORLD,&status);
  Solver( bufm, lenm, bufr, &lenr );
  free(bufm);
  c='D';
  MPI_Send(&c,1,MPI_CHAR,0,51,
                MPI_COMM_WORLD);
  MPI_Send(&z,1,MPI_INT,0,56,
                MPI_COMM_WORLD);
  MPI]_Send(&lenr,1,MPI_INT,0,57,
                MPI_COMM_WORLD);
  MPI_Send(bufr,lenr,MPI_CHAR,0,58,
                MPI_COMM_WORLD);
  free(bufr);
}
```

Each of the messages is additionally specified with its number for the accurate implementation of the communication protocol between master and workers, which represents two-level dynamic scheduling. The first level is applied for solving a system for clans, the second level is applied for parallel-sequential composition of clans.

5.2 Communication with Workers for Solving Systems on Clans

Communication with workers is regulated by the basic protocol (Fig. 5), which is developed into a more detailed scheme (represented with Figs. 6 and 7) with regard to a situation when there is no job for a worker—though it can be required further for parallel-sequential solving of composition systems. Overlapping the mentioned two levels is a direction for future work.

The protocol of master-worker communication specified by a Petri net and represented in Fig. 5 has been verified and optimized using the technique studied in [1]. It possesses the properties of an ideal communication protocol model: liveness, boundedness, and safeness. This means it can work for an unlimited time without deadlocks and overflow of buffers; each action can be implemented from any valid state.

To keep a few workers waiting for a job to appear, a waiting list is organized by the master. The waiting list consists of an array waiting_list and its size waiting_num. When there is no job for a ready worker, its process number is added to the list:

```
waiting_list[waiting_num++] = p;
```

When a worker is required, its process number is taken from the waiting list:

```
p = waiting_list[-waiting_num];
```

When solving systems for clans, there is no situation in which processes are taken from the waiting list. If the number of workers is greater than the number of clans, we start the required number of workers, and the rest enter the waiting list. If the number of workers is smaller than the number of clans, we start a job for a new clan immediately after receiving the "Ready" message. After solving the systems for all the clans, process numbers of ready workers stored in the waiting list are passed to the composition of clans.

Solving systems for clans on a given number of workers is represented by the following code:

```
zs = zf = zc = 0;
while(zs < nz || zf < nz || zc < numprocs-1) {
  MPI_Recv(&c,1,MPI_CHAR,MPI_ANY_SOURCE,51,
                MPI_COMM_WORLD,&status);
  p = status.MPI_SOURCE;
  switch(c) {
    case 'R': // job request
      if (zs >= nz) {
        waiting_list[waiting_num++] = p;
        zc++;
        break;
      }
      c1 = 'W';
      zs++;
      MPI_Send(&c1,1,MPI_CHAR,p,52,
                MPI_COMM_WORLD);
      MPI_Send(&zs,1,MPI_INT,p,53,
                MPI_COMM_WORLD);
      buf=system[zs].buf;
      len = system[zs].len;
      MPI_Send(&len,1,MPI_INT,p,54,
                MPI_COMM_WORLD);
      MPI_Send(buf,len,MPI_CHAR,p,55,
                MPI_COMM_WORLD);
      break;
    case 'D': // data
      MPI_Recv(&z,1,MPI_INT,p,56,
                MPI_COMM_WORLD,&status);
      MPI_Recv(&len,1,MPI_INT,p,57,
                MPI_COMM_WORLD,&status);
      buf = malloc(len);
      solution[z].len = len;
      solution[z].buf = buf;
      MPI_Recv(buf,len,MPI_CHAR,p,58,
                MPI_COMM_WORLD,&status);
      zf++;
      break;
  }
}
```

Here zs, zf, and zc are the numbers of started jobs, finished jobs, and closed (waiting) workers, respectively. After receiving message number 51, the master extracts the worker process number p from the message status and processes the message depending on its type. If the worker is ready ('R'), the processing depends on whether there is a job for that worker. In case there is no job ($zs \geq nz$, where nz is the total number of clans), the worker process number

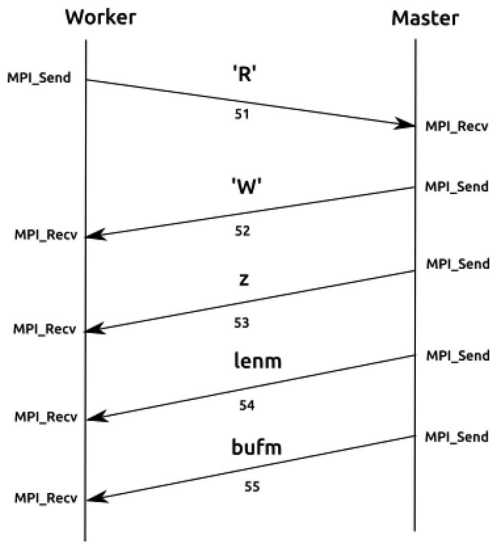

Fig. 6. Protocol of communicating master with workers for sending a system (a time diagram).

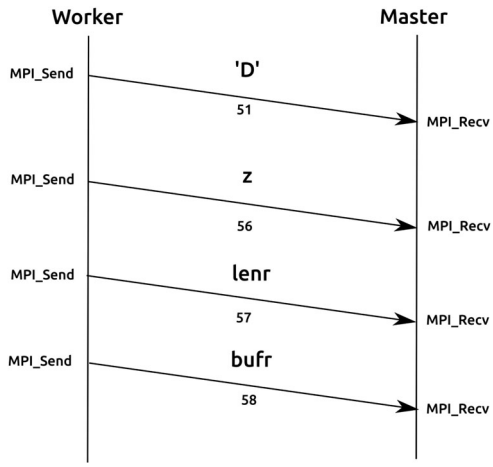

Fig. 7. Communication protocol between master and workers for sending results – basis solutions (a time diagram).

is added to the waiting list. Otherwise, a job is created and sent to the worker. We represent a generalized code using simple memory buffers as an analog to the sparse matrix files with two fields: len, the buffer length, and buf, the buffer address. All the systems are contained in an array called *system* while all the solutions are stored in an array called *solution*. It is expected that a worker will return the same system number as the one sent to the worker (z = zs).

5.3 Communication with Workers for Composition of Clans

Composition of clans is implemented either by simultaneously solving a single system or via parallel-sequential pairwise composition of clans; the corresponding choice is given with the command line flags "-c" or "-s", respectively. The simultaneous composition is implemented by the master after it first closes all workers in the waiting list:

```
if ( numprocs > 1) {
  for(p = 0;p < waiting_num;p++) {
    c1 = 'C';
    MPI_Send(&c1,1,MPI_CHAR,waiting_list[p],
                  52, MPI_COMM_WORLD);
  }
  waiting_num = 0;
}
```

Parallel-sequential composition is organized in terms of tasks that are dynamically scheduled for execution among the workers. The dynamic parallel-sequential collapse of the decomposition graph is provided by two functions: *ChooseEdge*, which chooses the next edge for the collapse, returning 1 when an edge has been chosen and 0 otherwise; and *CollapseEdge*, which implements a collapse (contraction) of a previously chosen edge. To coordinate the functioning of the two routines, two lists of vertices are kept: list 'e' of size n that specifies the actual current graph; and list 'pe' of size pn specifies a set of edges being currently collapsed (corresponding to systems that are being solved by workers). When an edge is chosen by ChooseEdge, it is removed from 'e' and inserted into 'pe'. For an edge choice ChooseEdge checks list 'pe' to avoid choosing an edge with vertices listed in 'pe'. In this way, processing of independent subsets of edges is provided. Before the choice, edges of 'e' are sorted in the order of descending edge weights for ensuring a greedy strategy of the edge choice.

The protocol of the master and workers' interaction for parallel-sequential composition of clans (Fig. 8) is the most

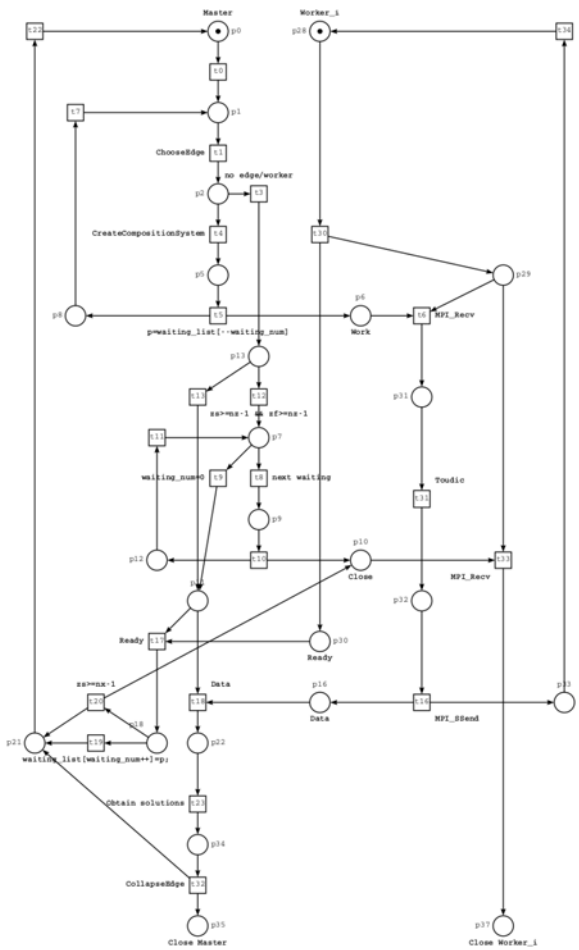

Fig. 8. Protocol of communicating master with workers for parallel-sequential composition (a Petri net).

complex because it takes into consideration the dynamic process when new jobs appear only after finishing previous ones. That is why taking workers from the waiting list specified by t_1, t_4, and t_5 is the basic way of dispatching jobs to workers. The central part of the communication protocol represented by t_8, t_{10} is closing workers when there are no more available jobs for them. The bottom part represents receiving and processing a message from a worker. After receiving data with t_{18} and processing it with t_{23} and t_{32}, the communication protocol returns to the main loop. The protocol represented in Fig. 8 has been verified and optimized using the technique studied in [1], considered with more detail in the previous subsection for the protocol shown in Fig. 5.

Parallel-sequential composition of clans on a given number of workers is represented by the following code:

```
while(zs < nz-1 || zf < nz-1 || zc < numprocs-1) {
  while(waiting_num > 0 && maybe_edge) {
    if(ChooseEdge('f', e, &n, pe, &pn)) {
      ce.v1 = pe[pn-1].v1; ce.v2 = pe[pn-1].v2;
      ce.w = pe[pn-1].w;
      z = ce.v1*(nx+1)+ce.v2;
      FilterContactPlaces(ce.v1,ce.v2,z);
      CreateCompositionSystem(ce.v1,ce.v2,z);
      c1 = 'W';
      zs++;
      p=waiting_list[-waiting_num];
      MPI_Send(&c1,1,MPI_CHAR,p,52,
                  MPI_COMM_WORLD);
      MPI_Send(&z,1,MPI_INT,p,53,
                  MPI_COMM_WORLD);
      len = system[z].len;
      buf = system[z].buf;
      MPI_Send(&len,1,MPI_INT,p,54,
                  MPI_COMM_WORLD);
      MPI_Send(buf,len,MPI_CHAR,p,55,
                  MPI_COMM_WORLD);
    } else maybe_edge=0;
  }
  if(zs >= nz-1 && zf >= nz-1) {
    for(p = 0;p < waiting_num;p++) {
      c1 = 'C';
      MPI_Send(&c1,1,MPI_CHAR,
        waiting_list[p],52,MPI_COMM_WORLD);
      zc++;
    }
    waiting_num = 0;
    if(zs < nz-1 && zf < nz-1 && zc < numprocs - 1)
      break;
  }
  MPI_Recv(&c,1,MPI_CHAR,MPI_ANY_SOURCE,51,
              MPI_COMM_WORLD,&status);
  p = status.MPI_SOURCE;
  switch(c) {
    case 'R':
      if(zs < nx - 1)
        waiting_list[waiting_num++] = p;
      else {
        c1='C';
        MPI_Send(&c1,1,MPI_CHAR,p,52,
              MPI_COMM_WORLD);
        zc++;
      }
      break;
    case 'D':
      MPI_Recv(&z,1,MPI_INT,p,56,
              MPI_COMM_WORLD,&status);
      ce.v2=z%(nx+1); ce.v2 = z%(nx+1);
      ce.w = 0;
      MPI_Recv(&len,1,MPI_INT,p,57,
              MPI_COMM_WORLD,&status);
      buf = malloc(len);
      solution[z].len = len;
      solution[z].buf = buf;
      MPI_Recv(buf,len,MPI_CHAR,p,58,
              MPI_COMM_WORLD,&status);
      zf++;
      AddUnitSolutions(&solution[z]);
      ComposeJointMatrix(&solution[ce.v1],
                         &solution[ce.v2]);
      MultiplySPM(&solution[ce.v1],&solution
      [z]);
      MinimizeBasis(&solution[ce.v1]);
      CollapseEdge(&ce, e, &n, pe, &pn);
      maybe_edge = 1;
      break;
    }
  }
```

The outer 'while' loop provides conditions rather similar to the ones described in the previous subsection. The difference is that the number of contracted edges for a connected graph is the number of vertices minus one. The nested 'while' loop implements the edge collapse when there is both a waiting worker process and a valid edge to be contracted. The variable maybe_edge enables it to avoid active waiting in the following way: it is reset when there are no edges to be contracted (namely when ChooseEdge returns 0) and it is set when such an edge can appear (namely after calling CollapseEdge).

When composing a job for a worker, the chosen edge (numbers of its vertices) is encoded into a variable 'z' and decoded from the received message when the worker returns its results. Routine FilterContactPlaces selects contact places for a chosen pair of connected clans (an edge). Routine CreateCompositionSystem creates a composition system for a chosen pair of clans and stores it in system [z] buffer.

When data are received from a worker, the contracted edge vertices are recovered from variable 'z' and stored into ce.v1 and ce.v2. For the edge encoding-decoding, a radix equal to the number of clans plus one is chosen that gives unique numbers of all the systems solved. Received basis solutions are stored in the buffer of solution[z], supplied with unit solutions for absent variables by AddUnitSolutions. Then, a joint matrix of basis solutions for a pair of clans is composed by ComposeJointMatrix and stored in the solution buffer of the first contracted clan solution[ce.v1]; multiplication of this matrix by the composition system solutions with MultiplySPM gives an intermediate result stored in the same buffer of solution[ce.v1]. Finally, the basis minimized with the MinimizeBasis routine is obtained. To accomplish the process, the edge is actually contracted by CollapseEdge.

TABLE 1
Specification of Selected Models for Benchmarks

Notation	Name	Places	Transitions	Arcs	Clans
al	AirplaneLD	7019	8008	30528	4
dc	DLCround	5343	8727	24849	3621
af	AutoFligh	3950	3936	9104	2833
cd	CloudDeployment	2271	19752	389666	1345
sm	SharedMemory	2651	5050	20000	51
ht	HypertorusGrid	2025	5184	20736	162

6 Analysis of Obtained Benchmarks

For obtaining benchmarks, Petri net models from the Model Check Contest (MCC) collection [26]—especially scalable ones that have a parametric specification—have been used. The following models have been selected: AirplaneLD, a simplified version of a landing detector for an airplane; DLCround, a distributed language compiler; AutoFlight, an automatic flight control system; CloudDeployment, a cloud application deployment; SharedMemory, processes sharing memory; HypertorusGrid, hypertorus communication grid. Specifications of the selected models' sizes and their ability to decompose are represented in Table 1. Note that random sparse matrices are mostly indivisible. To check the scalability of the clans composition, we also develop a dedicated generator of clan structure *gclans* with random values obtained according to a given range and density.

To run programs, we use two kinds of hardware: a desktop computer, Hare (Intel Core i5 3.2 GHz, 4 cores), and a cluster Saturn [27], including an Alembert cluster of nine double Intel Haswell E5-2650 v3 @ 2.30 GHz CPUs (10 cores each) and Descartes cluster having 16 nodes of two Gainestown E5520 @2.27 GHz (8 cores each). Computer Hare works under Ubuntu 16.04, gcc 5.4.1 from the GNU Compiler Collection with OpenMP 4.0 and libc 2.23, MPICH 3.2. Cluster Saturn [27] works under Scientific Linux 7.3; compute nodes are accessible through Slurm allocation. Also, we use GraphBLAS 1.1.2 from the SuiteSparse distribution for sparse matrix multiplication.

We first consider benchmarks of solving linear Diophantine systems in non-negative integers with ParTou on various numbers of cores on a desktop computer, Hare, and on a node of the Saturn Alembert cluster (20 cores). Then, we provide comparisons with a solver *struct* from the Tina toolbox [12], which is considered the best at the moment. Second, we estimate speedup of solving linear Diophantine systems via simultaneous and parallel-sequential composition of clans using ParAd with a solver *zsolve* from the 4ti2 toolbox [24] on nodes of the Saturn Descartes cluster. Note that most of the MCC models are decomposed into clans, which provides the opportunity for solving them faster using composition. A preliminary check of decomposition ability can be implemented with tool Deborah [21], developed in 2005.

TABLE 2
ParTou Benchmarks on a Desktop Hare
(1 Thread – Seconds, 2-16 Threads – Speedups)

Model: Threads	1	2	4	8	16
al	1368.116 s	1.51	2.92	3.09	3.33
dc	1445.034 s	1.66	2.41	2.29	2.31
af	257.868 s	1.46	1.78	1.62	1.67
cd	54.818 s	1.01	1.01	1.01	1.01
sm	178.438 s	1.64	2.54	2.16	2.15
ht	78.698 s	1.67	2.32	2.05	2.09

TABLE 3
ParTou Benchmarks on a Cluster Saturn Node
(1 Thread – Seconds, 4-40 Threads – Speedups)

Model: Threads	1	4	10	20	40
al	2179.910 s	4.09	7.47	9.37	11.74
dc	2520.872 s	3.19	5.87	6.21	3.23
af	366.799 s	2.42	3.17	2.44	2.5
cd	35.954 s	1.02	1.02	1.02	1.02
sm	291.491 s	3.41	4.76	1.69	0.93
ht	108.563 s	3.03	4.46	3.57	1.86

ParAd is launched from the command line that specifies input and output file names and options. The input file contains a given system matrix while the output file stores obtained basis solutions. An option "-c" specifies simultaneous composition of clans and an option "-s" specifies parallel-sequential composition of clans, by default no composition is applied. The solver name is specified with "-r name" option. The deafault solver ParTou provides an option "-c number" to specify the number of employed threads (cores), by default the maximal available number of threads is used.

Benchmarks obtained for solving systems with ParTou are primary because they reveal a certain speedup, even in cases where a system is indecomposable. Running times for various models on various numbers of threads (cores) obtained on the Hare computer and Saturn cluster are compared in Tables 2 and 3, respectively. Though we adjusted scaling parameters, there is certain amont of scatter in terms of times for various models. For instance, rather small values are obtained for a hypertorus model (4 dimension); using a model of 5 dimension gives about $100\times$ larger values.

We can see that the best time is achieved where the number of threads is close to the number of actual cores, and hyperthreading is not of much use except for a few separate cases. When the number of threads equals the actual number of cores, the obtained speedup is, on average, $0.5c$, where c is the number of cores. For some bigger models, higher speedup has been obtained—up to $0.75c$—for AirplaneLD with scaling parameter 2000; the corresponding run on a single core of a Saturn node takes about 5 hours.

A comparison of running times for various models obtained by ParTou and struct is represented by diagrams in Figs. 9 and 10 for the Hare computer and the Saturn cluster, respectively. One can conclude that even on a regular computer like Hare, ParTou runs about 4 times faster than one of the best programs (i.e., struct), while on a Saturn cluster node it runs about 8 times faster.

Decomposition into clans gives stable speedup when solving Diophantine systems in the integer domain. Possibilities of gaining speedups when solving a system in non-negative numbers is very rare, even when a system is decomposed in clans. This is easily explained by the fact that the space complexity of solving a system in non-negative integers is exponential, with rather big and unpredictable numbers of basis solutions. As far as each solution

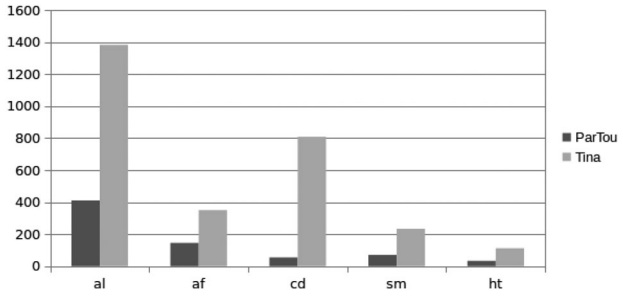

Fig. 9. Comparison of bencmarks for ParTou and struct on a desktop computer Hare (seconds).

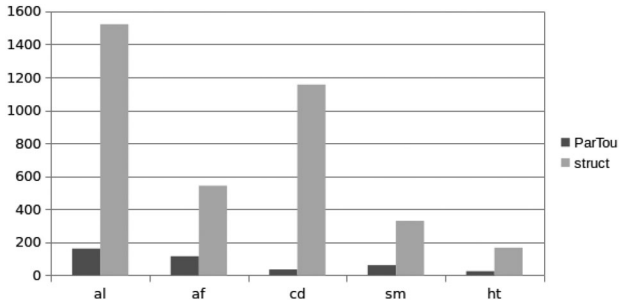

Fig. 10. Comparison of bencmarks for ParTou and struct on a cluster Saturn node (seconds).

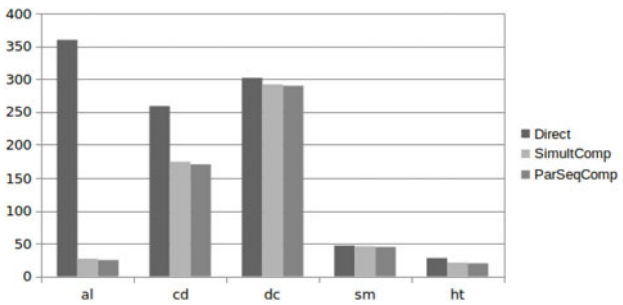

Fig. 11. ParAd benchmarks on a cluster Saturn (seconds).

should be stored when solving the final system of composition, some improvements are considered impossible.

Further, we study some benchmarks of running ParAd with zsolve to solve in parallel clans and parallel-sequential composition of clans in the integer domain. Note that even when solving a system on a single node, composition gives a certain speedup because it enables solving a sequence of systems of lesser dimension [13].

Benchmarks for composition of clans obtained on a few nodes of the Saturn cluster for various models are presented in Fig. 11. The number of nodes has been chosen in the range of 5–9 to run a separate clan on a separate core. The results demonstrate the fact that composition of clans is useful from a practical point of view, and it gives up to 10× speedup for models with moderate numbers of clans and 5–20 percent for models with larger numbers of clans. Sequential composition offers modest advantages in the present implementation, which can be explained by rather large numbers of little clans. For future implementations, some technique for aggregation of small clans seems useful.

Finally, we study the scalability of clan composition using a dedicated generator of clan structure, *gclans*. It creates an ideal matrix structure, which can be considered as a

TABLE 4
ParAd Scalability Benchmarks on a Cluster Saturn for Matrices Obtained by Gclans (1 Node – Seconds, 5-16 Nodes – Speedups)

Clans: Nodes	1	5	9	13	16
4×400	13.921 s	**2.39**	2.33	2.32	2.28
8×400	32.441 s	1.96	**2.38**	2.35	2.36
12×400	57.824 s	1.73	1.87	**2.06**	2.05
15×400	80.203 s	1.58	1.77	1.73	**1.88**
20×400	159.779 s	1.35	1.42	1.45	1.46
40×400	395.004 s	1.26	1.32	1.33	1.34

TABLE 5
ParAd Scalability Benchmarks on a Cluster Saturn for Real-Life Models (1 Node – Seconds, 4-16 Nodes – Speedups)

Model: Nodes	1	5	9	13	16
al-1000	46.477 s	1.29	1.28	1.27	1.27
al-2000	305.391 s	1.21	1.20	1.19	1.18
al-4000	2222.194 s	1.18	1.17	1.16	1.15
ht-4D	34.479 s	1.88	1.91	1.92	1.93
ht-5D	484.332 s	1.31	1.32	1.39	1.62

benchmark in the best case. To obtain a matrix, we specify the clan size, the number of clans, the values range, and the clans (connections) density. Solving a system consisting of k clans, we load k nodes at the first stage (Fig. 1), solving systems for all the clans in parallel. For a general Diophantine system, similar nodes work approximately the same time. Then, at the second stage, the composition system is solved on a single node. Thus, an ideal time of solving a system equals about double the time of solving a system for a clan. When the number of available nodes is smaller than the number of clans, the duration of the first stage is increased by the corresponding factor. The results collected into Table 4 confirm the above reasoning, illustrating the scalability of the technique: the best result for 4 clans is achieved on 5 nodes; for 9 clans on 10 nodes; and for 15 clans on 16 nodes. Note that Table 4 does not reflect the fact that the composition of clans itself gives about k^2 times speedup compared to the direct solution of a system where k is the number of clans.

Scalability of results for real-life models is represented in Table 5. For the first model AirplaneLD (al), the number of clans remains the same (4 clans) and the best performance is achieved for 5 computing nodes with about double speed-up. For the second model HypertorusGrid (ht), the number of clans grows rapidly with the scaling parameter corresponding to the number of dimensions. Greater number of computing nodes gives better results though it grows rather modestly because we do not have the required number of nodes (compared to the clans number shown in Table 1) in Decartes cluster and the composition system is rather big compared to the clan size. Note that for 6D case, 4ti2 [24] fails to solve the system ether directly or via simultaneous composition while parallel-sequential composition solves it in about two hours.

7 Conclusions

In this paper, we studied the implementation of ParAd, software for solving linear Diophantine systems on modern parallel architectures using OpenMP and MPI. Its theoretical

base is constituted by methods for solving linear systems via simultaneous and sequential composition of their clans [13] and utilizes a technique of parallel-sequential composition of clans, developed in this paper. For obtained benchmarks, a collection of Petri nets, i.e., sparse matrices, which represent models of real-life systems, has been used and increases the credibility of the obtained benchmarks.

ParAd represents an environment for solving systems using simultaneous and parallel-sequential composition of their clans, which can use various solvers for solving a linear system. ParAd runs on a cluster of compute nodes using MPI. A set of protocols for communicating master and worker processes has been specified by Petri nets and verified; a dynamic task dispatching subsystem has also been developed. The required number of nodes is limited by the number of clans a system is decomposed to. Benchmarks show that ParAd gives up to 10 times speedup.

Further, we developed ParTou, a solver of linear Diophantine systems in non-negative numbers. The performance has been improved compared to Adriana due to the development of a parallel algorithm and its implementation using OpenMP for running on multi-core architectures. The best obtained speedup on 20 cores is about 15 times.

Note that GPU issues [28], [29] are beyond the scope of this paper. Solving heterogeneous systems, balancing clan size, and using asynchronous MPI communication schemes including RMA (Remote Memory Access), represent directions for future work as well as fault-tolerance issues.

Acknowledgments

The authors thank Fullbright Scholar Program for the first author scholarship to visit the Innovative Computing Laboratory at the University of Tennessee, Knoxville, USA during autumn 2017. The authors also thank anonymous reviewers for their valuable comments which inspired directions for future work.

References

[1] D. A. Zaitsev, *Clans of Petri Nets: Verification of Protocols and Performance Evaluation of Networks*," Saarbrücken, Germany: LAP LAMBERT Academic Publishing, 2013, Art. no. 292.
[2] G. Xie, C. Li, Z. Dang, "Linear reachability problems and minimal solutions to linear diophantine equation systems," *Theoretical Comput. Sci.*, vol. 328, no. 1–2, pp. 203–219, 2004.
[3] J.-M. Champarnaud, J.-P. Dubernard, F. Guingne, H. Jeanne, "Geometrical regular languages and linear diophantine equations: The strongly connected case," *Theoretical Comput. Sci.*, vol. 449, pp. 54–63, 2012.
[4] G. A. Narboni, "From Prolog III to Prolog IV: The logic of constraint programming revisited," *Constraints*, vol. 4, no. 4, 2016, pp. 313–335.
[5] M. Alekhnovich, "Linear diophantine equations over polynomials and soft decoding of reed-solomon codes," *IEEE Trans. Inf. Theory*, vol. 51, no. 7, pp. 2257–2265, Jul. 2005.
[6] Q. F. Zhou, Q. T. Zhang, F. C. M. Lau, "Diophantine approach to blind interference alignment of homogeneous K-User 2x1 MISO broadcast channels," *IEEE J. Selected Areas Commun.*, vol. 31, no. 10, pp. 2141–2153, Oct. 2013.
[7] S. Martin, W. M. Brown, J.-L. Faulon, D. Weis, D. Visco, and J. Kenneke, "Inverse design of large molecules using linear diophantine equations," in *Proc. IEEE Comput. Syst. Bioinf. Conf.*, Aug. 2005, pp. 11–14.
[8] M. Jantzen, "Complexity of place/transition nets," in *Proc. Advances Petri Nets*, 1986, pp. 413–434.
[9] S. L. Kryvyi, "An algorithm for constructing the basis of the solution set for systems of linear diophantine equations over the ring of integers," *Cybern. Syst. Anal.*, vol. 45, pp. 875–880, 2009.
[10] H. Alaivan, J. M. Toudic, "Recherche des semi-flots des versous et des trappes dans les resaux de Petri," *Technique Sci. Informatiques*, vol. 4, no. 1, pp. 103–112, 1985.
[11] J. M. Colom, M. Silva, "Convex geometry and semiflows in P/T nets: A comparative study of algorithms for computation of minimal p-semiflows, in *Proc. Advances Petri Nets*, 1990, pp. 79–112.
[12] B. Berthomieu, P.-O. Ribet, and F. Vernadat, "The tool TINA: Construction of abstract state spaces for petri nets and time petri nets," *Int. J. Production Res.*, vol. 42, no. 14, pp. 2741–2756.
[13] D. A. Zaitsev, "Sequential composition of linear systems' clans," *Inf. Sci.*, vol. 363, 2016, pp. 292–307.
[14] *The Sourcebook of Parallel Computing*, Ed. J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, A. White, Amsterdam, The Netherlands: Elsevier, Oct. 2002.
[15] T.A. Davis, *Direct methods for sparse linear systems*, Philadelphia, PA, USA: SIAM, Sep. 2006.
[16] Message Passing Interface Forum, "MPI: A message passing interface standard," (2015). [Online]. Available: http://www.mpi-forum.org/
[17] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*, Boston, MA, USA: MIT Press, 1996.
[18] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*," Cambridge, MA, USA: MIT Press, 1994.
[19] OpenMP Architecture Review Board: OpenMP Application Program Interface Version 3.0, (2008). [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf
[20] B. Chapman, G. Jost, and R. Pas, *Using OpenMP Portable Shared Memory Parallel Programming*, Cambridge, MA, USA: MIT Press, 2008.
[21] D. A. Zaitsev, "Decomposition of Petri nets," *Cybernetics Syst. Anal.*, vol. 40, no. 5, pp. 739–746, 2004.
[22] D. A. Zaitsev, "Compositional analysis of Petri nets," *Cybernetics and Systems Analysis*, vol. 42, no. 1, pp. 126–136, 2006.
[23] R. Merris, *Graph Theory*, Hoboken, NJ, USA: Wiley, 2011.
[24] 4ti2 – A software package for algebraic, geometric and combinatorial problems on linear spaces. (2018). [Online]. Available: https://4ti2.github.io/
[25] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.
[26] Model Checking Contest, (2018). [Online]. Available: https://mcc.lip6.fr/models.php
[27] The Saturn Cluster, (2018). [Online]. Available: https://bitbucket.org/icl/saturn-users
[28] GPU Computing and Applications, Ed. Y. Cai, S. See, Berlin, Germany: Springer, 2015.
[29] S. Tomov, J. Dongarra, M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parellel Comput. Syst. Appl.*, vol. 36, no. 5–6, pp. 232–240, 2010.

Dmitry Zaitsev (M'10–SM'11) received the Eng. degree in applied mathematics from Donetsk Polytechnic Institute, Donetsk, Ukraine, in 1986, the PhD degree in automated control from the Kiev Institute of Cybernetics, Kiev, Ukraine, in 1991, and the Dr.Sc. degree in telecommunications from the Odessa National Academy of Telecommunications, Odessa, Ukraine, in 2006. He is a professor of Computer Engineering at International Humanitarian University, Odessa, Ukraine since 2009 and also a professor of Computer Science at Vistula University, Warsaw, Poland since 2014. He developed the analysis of infinite Petri nets with regular structure, the decomposition of Petri nets in clans, and the method of synthesis of fuzzy logic function given by tables. His current research interests include Petri net theory and its application in networking, production control, and computing. Recently he has been a co-director in Austria-Ukraine, China-Ukraine, and Slovakia-Ukraine research projects. He developed small universal Petri nets, the analysis of infinite Petri nets with regular structure, the decomposition of Petri nets in clans, and the method of synthesis of fuzzy logic function given by tables. He designed Opera-Topaz system for production control, models of protocols and networking technologies TCP, BGP, IOTP, MPLS, Bluetooth, PBB, offered and implemented in Linux kernel a new stack of networking protocols E6. He developed the analysis of infinite Petri nets with regular structure, the decomposition of Petri nets in clans, and the method of synthesis of fuzzy logic function given by tables. His current research interests include Petri net theory and its application in networking, production control, and computing. He is a senior member of the ACM and IEEE.

Stanimire Tomov received the master of science degree in computer science from Sofia University, St. Kliment Ohridski, Bulgaria, in 1994, and the PhD degree in mathematics from Texas A&M University, in 2002. He worked at the Brookhaven National Laboratory before joining ICL in 2004. Stanimire (Stan) Tomov, PhD, is a research director in the Innovative Computing Laboratory (ICL) and research assistant professor in the Electrical Engineering and Computer Science Department, University of Tennessee, Knoxville. His research interests include parallel algorithms, numerical analysis, and high-performance scientific computing (HPC). He has been involved in the development of numerical algorithms and software tools in a variety of fields ranging from scientific visualization and data mining to accurate and efficient numerical solution of PDEs. Currently, his work is concentrated on the development of numerical linear algebra libraries for emerging architectures for HPC, such as heterogeneous multicore processors, graphics processing units (GPUs), and Many Integrated Core (MIC) architectures. In particular, he is leading the development of the Matrix Algebra on GPU and Multicore Architectures (MAGMA) libraries, targeting to provide LAPACK/ScaLAPACK functionality on the next-generation of architectures. Tomov is also a Principal Investigator of the CUDA Center of Excellence (CCOE) at UTK, and Co-PI of the Intel Parallel Computing Center (IPCC) at ICL.

Jack Dongarra received a bachelor of science degree in mathematics from Chicago State University, in 1972, the master of science degree in computer science from the Illinois Institute of Technology, in 1973, and the PhD degree in applied mathematics from the University of New Mexico, in 1980. He worked at the Argonne National Laboratory until 1989, becoming a senior scientist. He now holds an appointment as University Distinguished professor of Computer Science in the Electrical Engineering and Computer Science Department, University of Tennessee and holds the title of Distinguished Research Staff in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL); Turing fellow at Manchester University; an adjunct professor in the Computer Science Department, Rice University; and a faculty fellow of the Texas A&M University's Institute for Advanced Study. He is the director of the Innovative Computing Laboratory at the University of Tennessee. He is also the director of the Center for Information Technology Research, University of Tennessee which coordinates and facilitates IT research efforts at the University. He specializes in numerical algorithms in linear algebra, parallel computing, the use of advanced-computer architectures, programming methodology, and tools for parallel computers. His research includes the development, testing and documentation of high quality mathematical software. He has contributed to the design and implementation of the following open source software packages and systems: EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS, and PAPI. He has published approximately 200 articles, papers, reports and technical memoranda and he is coauthor of several books. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high performance computers using innovative approaches; in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement; in 2011 he was the recipient of the IEEE Charles Babbage Award; and in 2013 he was the recipient of the ACM/IEEE Ken Kennedy Award for his leadership in designing and promoting standards for mathematical software used to solve numerical problems common to high performance computing. He is a fellow of the AAAS, ACM, IEEE, and SIAM and a foreign member of the Russian Academy of Sciences and a member of the US National Academy of Engineering.