arXiv:2110.08537v1 [cs.LO] 16 Oct 2021

Verification of MPI programs

Andrew M. Mironov

Moscow State University, Faculty of Mechanics and Mathematics
amironov66@gmail.com

Abstract. In this paper, we outline an approach to verifying parallel programs. A new mathematical model of parallel programs is introduced. The introduced model is illustrated by the verification of the matrix multiplication MPI program.

1 Introduction

Parallel programs are computer programs designed to be executed on multiprocessor computing systems (MPCS). The problem of developing correct and safe parallel programs is currently highly topical. Formal verification of correctness and safety properties of parallel programs is a complex mathematical problem. The existing methods for solving this problem are suitable only for a limited class of parallel programs.

One of the most widely used languages for describing parallel programs is MPI (Message Passing Interface).

In this paper, a new mathematical model of MPI programs is introduced. On the basis of this model one can solve the problems of verifying parallel programs presented on a certain subset of MPI. The introduced model is illustrated by the verification of the matrix multiplication MPI program.

The most essential feature of the approach to modeling and verification of MPI programs presented in this paper is the possibility of using this approach for MPI programs that can generate any number of processes.

Among other approaches to modeling and verifying MPI programs for any number of processes it should be noted the approach in the work [1]. In this work the tool for modeling and verifying MPI programs called ParTypes is presented. It requires the user to provide a protocol which specifies the communication pattern of an execution. Unfortunately, it has some limitations. In particular, it does not work for wildcard receives, which means it cannot be applied to the MPI program for matrix multiplication considered in the present paper. There are other approaches using symbolic execution and model checking ([3]-[10]), but all of them require a bound on the number of processes.

2 Essentials for MPI

2.1 MPI programs

MPI (Message Passing Interface) is a set of functions, types and constants for a development of parallel programs (called **MPI programs**). A MPI program

is a C program in which functions, types and constants from MPI can be used. An execution of an MPI program on a MPCS has the following form: at each node of the MPCS, a computational process corresponding to this MPI program is generated. All processes generated by the MPI program operate in parallel and can exchange information with each other through **message passing**.

Each process generated by a MPI program has a **rank**, which is a number from the set $\{0, \ldots, m-1\}$, where m is the number of processes generated by the MPI program. A process with rank 0 is called a **root process**.

MPI has functions `MPI_Comm_rank` and `MPI_Comm_size`. Each process generated by a MPI program can use these functions to find out its rank and a number of processes generated by this MPI program, respectively:

- `MPI_Comm_rank (MPI_COMM_WORLD, &rank);`
  after executing this function, a value of variable `rank` (`int`) will be equal to the rank of the process that called this function,
- `MPI_Comm_size (MPI_COMM_WORLD, &nprocs);`
  after executing this function, a value of variable `nprocs` (`int`) will be equal to the number of processes generated by the MPI program.

2.2 Message Passing Functions

In MPI, a **message** is an array of data of a certain type, and **message passing (MP)** is an action, as a result of which a message is sent by one process and is received by other process (or processes). A sent message is placed in a queue, from which it will then be taken by the receiving process.

We will consider the following types of MP MPI functions:

- **pairwise MP (PMP)**: there are two processes involved in a PMP: a sender of a message, and a receiver of this message,
- **broadcast MP (BMP)**: all processes generated by a MPI program participate in an executing of a BMP, root process is a sender of a message, and other processes are receivers of this message.

Messages sent by PMP functions cannot be received by BMP functions, and vice versa. Below we describe some of MP functions. In the descriptions, for each argument of these functions, we indicate its type in parentheses.

1. Sending a message (PMP) :

$$\texttt{MPI\_Send}\ (p, n, \tau, r, l, \texttt{MPI\_COMM\_WORLD}); \tag{1}$$

   This function is performed by sending a message to a process with rank r (`int`). The message being sent is an array of n (`int`) elements of type τ (`MPI_Datatype`), the beginning of which is at p (`void *`). The tag (i.e. label) l (`int`) is appended to the message being sent.

2. Receiving a message (PMP):

$$\texttt{MPI\_Recv}\ (p, n, \tau, \texttt{MPI\_ANY\_SOURCE}, \texttt{MPI\_ANY\_TAG}, \texttt{MPI\_COMM\_WORLD}, q); \quad (2)$$

This function is performed by receiving a message, which should be placed in the memory location the beginning of which is at p. It is assumed that the received message is an array of no more than n elements of type τ.
q `(MPI_Status *)` is an address of a structure in which information about the received message should be placed. This structure contains the following fields: `MPI_SOURCE` (the sender's rank must be placed in it), `MPI_TAG` (the received message tag must be placed in it), and other fields.

3. Sending a message from a root process to other processes (BMP):

$$\texttt{MPI\_Bcast}\ (p, n, \tau, 0, \texttt{MPI\_COMM\_WORLD}); \quad (3)$$

This function is performed as follows.
   - In a root process, a message is sent to all other processes, which is an array of n (`int`) elements of type τ (`MPI_Datatype`), the beginning of which is located at p (`void *`).
   - In other processes, a message is received from the root process, which must be located in the memory location at p. It is assumed that the received message is an array of no more than n elements of type τ.

3 Matrix Multiplication MPI Program

In this section, we present an example of a matrix multiplication MPI program. The example is taken from [2]. This example is used below to illustrate the application of the model of MPI programs described in this work for verifying MPI programs.

3.1 Informal Description of Matrix Multiplication MPI Program

The problem of matrix multiplication is to calculate a product $C = AB$ given the matrices A and B. Informally, the work of the MPI program Π for multiplying the matrices A and B, stated in this section, can be described as follows. We will call a root process of Π a **manager** and other processes of Π **workers**. Manager's job consists of the following actions:

- sending second matrix (B) to all workers,
- assigning tasks to workers, and receiving results from workers.

Each task for a worker is to calculate one row of the matrix $C = AB$. The manager assigns this task by sending the worker a message containing one row of matrix A. A tag of this message is equal to the number of the row being sent. As soon as the manager receives a result from a worker (i.e., a message with the calculated row of the product, its tag is equal to the number of this row), he sends this worker either a new task (if there are still unassigned tasks), or a message with tag 0 (if there are no unassigned tasks).

3.2 MPI Matrix Multiplication Program

The following matrix multiplication MPI program Π uses auxiliary function `vecmat` to multiply the row `vector[L]` by the matrix `matrix[L][M]` and write the result to the array `result[M]`. This function looks like this:

```
void vecmat (double vector[L],
             double matrix[L][M],
             double result[M])
{ int j, k;
  for (j = 0; j < M; j++)
    for (k = 0, result[j] = 0.0; k < L; k++)
      result[j] += vector[k]*matrix[k][j];
}
```

In the MPI program Π presented below, we use the following notation: `input(a, b);` and `output(c);` are abbreviations of the functions for reading from the file of the factors and writing to the file of the product, respectively.

A MPI program Π for multiplying matrices A and B has the following form (in this program, to the left of each line, we indicate its number, this is necessary to describe the correspondence between components of this program and components of the model $\mathcal{P}_{\Pi}$ of this program):

```
01 #define comm MPI_COMM_WORLD
02
03 int main(int argc, char *argv[])
04 { int rank, nprocs, i, j;
05   MPI_Status status;
06
07   MPI_Init(&argc, &argv);
08   MPI_Comm_size(comm, &nprocs);
09   MPI_Comm_rank(comm, &rank);
10
11   if (rank == 0)
12   { int count;
13     double a[N][L], b[L][M], c[N][M], tmp[M];
14
15     input(a, b);
16     MPI_Bcast(b, L*M, MPI_DOUBLE,
17               0, comm);
18     for (count = 0;
19          count < nprocs-1 && count < N;
20          count++)
21       MPI_Send(&a[count][0], L, MPI_DOUBLE,
22                count+1, count+1, comm);
23     for (i = 0; i < N; i++)
24     { MPI_Recv(tmp, M, MPI_DOUBLE,
```

```
25                MPI_ANY_SOURCE, MPI_ANY_TAG,
26                comm, &status );
27        for (j = 0; j < M; j++)
28          c[status.MPI_TAG-1][j] = tmp[j];
29        if (count < N)
30        { MPI_Send(&a[count][0], L, MPI_DOUBLE,
31                   status.MPI_SOURCE, count+1, comm);
32          count++;
33        }
34      }
35      for (i = 1; i < nprocs; i++)
36        MPI_Send(NULL, 0, MPI_INT,
37                 i, 0, comm);
38      output (c);
39    }
40
41    else
42    { double b[L][M], in[L], out[M];
43
44      MPI_Bcast(b, L*M, MPI_DOUBLE,
45                0, comm);
46      while (1)
47      { MPI_Recv(in, L, MPI_DOUBLE,
48                 0, MPI_ANY_TAG,
49                 comm, &status);
50        if (status.MPI_TAG == 0) break;
51        vecmat(in, b, out);
52        MPI_Send(out, M, MPI_DOUBLE,
53                 0, status.MPI_TAG, comm);
54      }
55    }
56
57    MPI_Finalize();
58    return 0;
59  }
```

3.3 Auxiliary notation

For the convenience of modeling and verification this program, we introduce special designations for some of the objects used in it:

- arrays `a[N][L]`, `b[L][M]`, `c[N][M]`, will be denoted by symbols A, B, C and interpreted as corresponding matrices,
- the message sent by the `MPI_Send` functions in lines 21, 22 and 30, 31 of the program will be understood as the corresponding row of the matrix A, and denoted by A_i, where $i = \texttt{count} + 1$,

– array `c[status.MPI_TAG-1][M]`, into which the message received by the `MPI_Recv` function in lines 24, 25, 26 is copied, we interpret it as a corresponding row of C, and denote it by C_l, where $l =$ `status.MPI_TAG`,
– from the definition of the `vecmat` function it follows that array `out` calculated as a result of the execution of the `vecmat` function on line 51 corresponds to the product of the row Y corresponding to array `in` by matrix B, we will denote this array `out` in the model of Π by the product YB.

3.4 A specification of the matrix multiplication MPI program

A specification of the MPI program Π is the following statement: after a completion of any execution of Π the equality $C = AB$ holds, i.e.

$$\forall\, i = 1, \ldots, N \quad C_i = A_i B. \tag{4}$$

4 A model of a MPI program

In this section, we introduce a concept of a model of a MPI program. This model is designed for formal representation of MPI programs that use the above message passing functions. The basic concepts of this model are sequential and distributed processes. A sequential process is a model of a computational process generated by a MPI program on a node of a MPCS, and a distributed process is a model of a MPI program on the whole. The proposed model is a theoretical basis for solving problems of verifying MPI programs.

4.1 Auxiliary concepts

We assume that there are given sets $\mathcal{T}$, $\mathcal{X}$ and $\mathcal{F}$, elements of which are called **types**, **variables**, and **function symbols (FS)**, respectively. Each element x of $\mathcal{X}$ and $\mathcal{F}$ is associated with some type $\tau_x \in \mathcal{T}$. $\forall\, f \in \mathcal{F}$ τ_f has the form

$$(\tau_1, \ldots, \tau_n) \to \tau, \quad \text{where } \tau_1, \ldots, \tau_n, \tau \in \mathcal{T}. \tag{5}$$

$\forall\, \tau \in \mathcal{T}$ a set $\mathcal{D}_\tau$ of **values** of type τ is given. $\mathcal{D}$ denotes a set of values of all types. $\mathcal{T}$ has the types $\mathbf{B}$, $\mathbf{N}$, $\mathbf{C}$, where $\mathcal{D}_{\mathbf{B}} = \{0, 1\}$, $\mathcal{D}_{\mathbf{N}}$ is the set of natural numbers $\{0, 1, \ldots\}$, values of type $\mathbf{C}$ are called **channels**.

$\forall\, f \in \mathcal{F}$, if τ_f has form (5), then this FS is associated with a function (denoted by the same symbol f) of the form $\mathcal{D}_{\tau_1} \times \ldots \times \mathcal{D}_{\tau_n} \to \mathcal{D}_\tau$.

The set $\mathcal{E}$ of **terms** is defined inductively. Each term $e \in \mathcal{E}$ is associated with a type $\tau_e \in \mathcal{T}$. The definition of a term is as follows: each $e \in \mathcal{D} \cup \mathcal{X}$ is a term of the type τ_e, and if $f \in \mathcal{F}$, $e_1, \ldots, e_n \in \mathcal{E}$, and $\tau_f = (\tau_{e_1}, \ldots, \tau_{e_n}) \to \tau$, then $f(e_1, \ldots, e_n)$ is a term of type τ.

$\forall\, e \in \mathcal{E}$ $\mathcal{X}_e = \{x \in \mathcal{X} \mid x \text{ occurrs in } e\}$, $\mathcal{E}_0 = \{e \in \mathcal{E} \mid \mathcal{X}_e = \emptyset\}$. Each $e \in \mathcal{E}_0$ is associated with a **value** $value(e) \in \mathcal{D}$, where $\forall\, e \in \mathcal{D}$ $value(e) = e$, and if $e = f(e_1, \ldots, e_n) \in \mathcal{E}_0$, then $value(e) = f(value(e_1), \ldots, value(e_n))$. $\forall\, e \in \mathcal{E}_0$ the value $value(e)$ will be denoted by the same notation e.

We will assume that

- $\forall\, n \geq 1$ $\mathcal{F}$ has FS $tuple_n$, which allows to construct tuples: for each list of terms $e_1, \ldots, e_n$, the set $\mathcal{E}$ has the term $tuple_n(e_1, \ldots, e_n)$, which we will denote by $(e_1, \ldots, e_n)$ and interpret as a tuple of terms $e_1, \ldots, e_n$,
- $\mathcal{F}$ has FS $channel$ of the type $\mathbf{N} \to \mathbf{C}$, $\forall\, i \geq 0$ the channel $channel(i)$ is said to be an i-th channel, a term of the form $channel(e)$ will be denoted by c_e,
- $\mathcal{D}_{\mathbf{C}}$ has a **broadcast channel** $\circ$, it differs from all channels c_i $(i \geq 0)$.

$\forall\, E \subseteq \mathcal{E}\ \ \forall\, \tau \in \mathcal{T}\ \ E_\tau = \{e \in E \mid \tau_e = \tau\}$. The sets $\mathcal{E}_{\mathbf{B}}$ and $\mathcal{E}_{\mathbf{C}}$ are denoted by $\mathcal{B}$ and $\mathcal{C}$, elements of $\mathcal{B}$ are called **formulas**. $\forall\, X \subseteq \mathcal{X}\ \ \mathcal{E}(X) = \{e \in \mathcal{E} \mid \mathcal{X}_e \subseteq X\}$, $\mathcal{B}(X) = \mathcal{E}(X) \cap \mathcal{B}$. Below, for each function $f : E \to E'$ under consideration, where $E, E' \subseteq \mathcal{E}$, we assume that $\forall\, e \in E\ \ \tau_{f(e)} = \tau_e$.

$\mathcal{D}^*$ denotes the set of all tuples of the form $(d_1, \ldots, d_n)$, where $n \geq 0$ and $d_1, \ldots, d_n \in \mathcal{D}$, if $n = 0$ then the corresponding tuple is said to be **empty** and is denoted by ε. Elements of $\mathcal{D}^*$ are called **queues**. $\forall\, D \in \mathcal{D}^*$ $|D|$ denotes the number of components in D. If $D \in \mathcal{D}^*$ and $1 \leq i \leq |D|$, then D_i denotes i-th component of D. $\forall\, M \subseteq \mathcal{D}^*$, $\forall\, i \geq 1$ M_i denotes the set of i-th components of tuples from M. If $D = (d_1, \ldots, d_n) \in \mathcal{D}^* \setminus \{\varepsilon\}$, then $head(D)$ and $tail(D)$ denote the value d_1 and the queue $(d_2, \ldots, d_n)$ respectively.

A **binding** is a function $\theta : \mathcal{X} \to \mathcal{E}$. The set of all bindings is denoted by Θ. We will use the following notation:

- $\forall\, X \subseteq \mathcal{X}\quad \Theta(X) = \{\theta \in \Theta \mid \forall\, x \in \mathcal{X} \setminus X\ \ \theta(x) = x\}$,
- $\forall\, \theta \in \Theta$, $\forall\, e \in \mathcal{E}\ \ e^\theta$ is a term obtained from e by replacing $\forall\, x \in \mathcal{X}_e$ each occurrence of x in e on the term $\theta(x)$,
- $\forall\, \theta, \theta' \in \Theta\ \ \theta\theta'$ is a binding such that $\forall\, x \in \mathcal{X}\ \ (\theta\theta')(x) = (x^\theta)^{\theta'}$.

4.2 Sequential processes

In this section we define a concept of a sequential process (SP). A SP is a model of a computational process generated by a MPI program on a node of a MPCS.

Elementary actions (EA) are notations of the following forms:

$$c!e, \quad c?e, \quad e := e', \quad [\![\varphi]\!], \quad \text{where } c \in \mathcal{C},\ e, e' \in \mathcal{E}, \tau_e = \tau_{e'}, \varphi \in \mathcal{B},$$

which are called a **sending** message e to channel c, a **receiving** message e from channel c, an **assignment**, and a **conditional transition**, respectively.

An **action** is a finite sequence of EAs, in which there is no more than one sending or receiving. An action α is called a **sending** or a **receiving** if one of EAs occurred in α is a sending or a receiving, respectively. An action that is not a sending or a receiving is called an **internal action**. Each EA can be considered as an action consisting of this EA. The set of all actions is denoted by $\mathcal{A}$. $\forall\, \alpha \in \mathcal{A}\ \ \mathcal{X}_\alpha$ is the set of all variables occurred in α. $\forall\, \theta \in \Theta, \forall\, \alpha \in \mathcal{A}\ \ \alpha^\theta$ denotes an action obtained from α by replacing each $x \in \mathcal{X}_\alpha$ on x^θ.

A **sequential process (SP)** is a triple (P, X, φ), components of which have the following meaning:

- P is a graph with a selected node P^0 (called an **initial node**), each edge of which has a label $\alpha \in \mathcal{A}$,

– $X \subseteq \mathcal{X}$ is a set **of input variables** of SP P, and
– $\varphi \in \mathcal{B}$ is an **initial condition** of SP P.

For each SP (P, X, φ)

– this SP is denoted by the same symbol P as a graph of this SP, the set of nodes of the graph P is also denoted by P,
– X_P and φ_P denote the second and third components of P respectively,
– $\mathcal{X}_P$ is the set of all variables occurred in P,
– $\hat{X}_P$ denotes the set $\mathcal{X}_P \setminus X_P$ of **private** variables of SP P,
– $\mathcal{A}_P$ denotes the set of labels of edges of P,
– $P^{v \to v'}$ denotes an edge of P from v to v'.
– each node of the graph P is an element of the set $\mathcal{D}$,
– P contains private variable at_P, and for each edge of the graph P, if v and v' are the start and the end of this edge, respectively, then the first EA in the label of this edge has the form $[\![at_P = v]\!]$, and the last one is $at_P := v'$, these EAs will not be specified explicitly.

A SP is a formal description of a behavior of a dynamic system, a work of which is a sequential execution of actions.

A **state** of SP P is a pair $s = (\theta^s, \{[c]^s \mid c \in \mathcal{D}_{\mathbf{C}}\})$, where

– $\theta^s \in \Theta(\mathcal{X}_P)$ is a binding, such that $\forall\, x \in \mathcal{X}_P \;\; \theta^s(x) \in \mathcal{E}_0$,
– $\forall\, c \in \mathcal{D}_{\mathbf{C}} \;\; [c]^s \in \mathcal{D}^*$ is a queue called a **content** of channel c in state s.

The set of all states of SP P is denoted by Σ_P.

$\forall\, s \in \Sigma_P$, $\forall\, e \in \mathcal{E}(\mathcal{X}_P)$, the value e^{θ^s} is denoted by e^s.

A state $s \in \Sigma_P$ is said to be **initial**, if $s = (\theta, \{\varepsilon \mid c \in \mathcal{D}_{\mathbf{C}}\})$, where $\varphi_P^\theta = 1$ and $at_P^s = P^0$. An initial state of P is denoted by 0_P. A state s of SP P is said to be **terminal** if there is no an edge outgoing from at_P^s.

Below we define a concept of a transition of a SP P corresponding to some action $\alpha \in \mathcal{A}_P$. This transition is a pair (s, s') of states from Σ_P, the relationship between s and s' can be understood as follows: if P is in the state s at the current time, then after sequential execution of EAs from α, a state of P will be s'.

$\forall\, s, s' \in \Sigma_P, \forall\, \alpha \in \mathcal{A}_P$ the notation $s \xrightarrow{\alpha} s'$ denotes the statement that (s, s') is a transition corresponding to α. This statement holds if α is a sequence of EAs of the form $\alpha_1 \dots \alpha_n$, and

$$\exists\, s_1, \dots, s_{n-1} \in \Sigma_P : s \xrightarrow{\alpha_1} s_1,\; s_1 \xrightarrow{\alpha_2} s_2,\; \dots,\; s_{n-1} \xrightarrow{\alpha_n} s',$$

where the statement $s \xrightarrow{\alpha} s'$ in the case when α is an EA is defined separately for each form of α, after the formal definition of this statement for each specific form of α we informally interpret the state change as a result of this transition:

(a) if $\alpha = c!e$, then $\theta^{s'} = \theta^s$, and

$$[c^s]^{s'} = ([c^s]^s, e^s), \;\; \forall\, c' \in \mathcal{D}_{\mathbf{C}} \setminus \{c^s\} \;\; [c']^{s'} = [c']^s,$$

in this case, P sends the value e^s to the channel c^s, after which

- the content of the channel c^s has been increased by adding e^s,
- the binding of variables from $\mathcal{X}_P$ did not change, and contents of all channels, except for the c^s channel, also did not change,

(b) if $\alpha = c?e$, then $[c^s]^s \neq \emptyset$ and

$$\exists\, \theta \in \Theta(\hat{X}_P) : (e^\theta)^s = head([c^s]^s), \theta^{s'} = \theta\theta^s,$$
$$[c^s]^{s'} = tail([c^s]^s), \quad \forall\, c' \in \mathcal{D}_{\mathbf{C}} \setminus \{c^s\} \;\; [c']^{s'} = [c']^s$$

in this case, P takes the value $head([c^s]^s)$ from the channel c^s and changes the current binding so that the value of the term e coincides with the accepted value on a new binding, after which

- the content of the channel c^s becomes $tail([c^s]^s)$,
- contents of the other channels have not changed,

(c) if $\alpha = (e := e')$, then $\begin{cases} \exists\, \theta \in \Theta(\hat{X}_P) : (e^\theta)^s = (e')^s, \theta^{s'} = \theta\theta^s, \\ \forall\, c \in \mathcal{D}_{\mathbf{C}} \;\; [c]^{s'} = [c]^s, \end{cases}$
in this case, P changes the current binding so that the value of the term e on the new binding would be equal to the value of the e' term on the old binding, contents of the channels does not change,

(d) if $\alpha = [\![\varphi]\!]$, then $\varphi^s = 1$, $\theta^{s'} = \theta^s$, and $\forall\, c \in \mathcal{D}_{\mathbf{C}} \;\; [c]^{s'} = [c]^s$,
in this case, the binding and contents of the channels are not changed.

An empty transition of SP P is a pair (s, s') of states from Σ_P, such that $\theta^s = \theta^{s'}$. The empty transition is denoted by $s \to s'$.

If a pair (s, s') is a transition of a SP, then we say that this is a transition from s to s', s and s' is called a **start** and an **end** of this transition, respectively.

Let P be a SP, and $v \in P$, $v \neq P^0$. If sets of all edges of P ending in v and starting at v are $\{v_i \xrightarrow{\alpha_i} v \mid i = 1, \ldots, n\}$ and $\{v \xrightarrow{\alpha'_i} v'_i \mid i = 1, \ldots, n'\}$ respectively, where v differs from all the nodes v_i and v'_i, and either all actions $\alpha_1, \ldots, \alpha_n$ are internal, or all actions $\alpha'_1, \ldots, \alpha'_{n'}$ are internal, then a **reduction** operation can be applied to P, which consists of transforming this graph by

- removing the node v and associated edges, and
- adding edges of the form $v_i \xrightarrow{\alpha_i \alpha'_{i'}} v'_{i'}$, where $i = 1, \ldots, n, i' = 1, \ldots, n'$, and $\alpha_i \alpha'_{i'}$ is a concatenation of sequences α_i and $\alpha'_{i'}$.

A **renaming** is an injective function $\eta : X \to X'$, where $X, X' \subseteq \mathcal{X}$. For each renaming $\eta : X \to X'$, each $e \in \mathcal{E}$ and each SP P, the notations e^η and P^η denote a term or a SP respectively, obtained from e or P by replacing $\forall\, x \in X$ of each occurrence of x by $\eta(x)$. If P is a SP, and η is a renaming of the form $\eta : \hat{X}_P \to \mathcal{X} \setminus X_P$, then we will consider SP P and P^η as equal.

4.3 Distributed Processes

In this section we introduce a concept of a distributed process, which can be used for formal representation of MPI programs.

A **distributed process (DP)** is a family of SPs $\mathcal{P} = \{P_i \mid i \in I\}$, where components of $\{\hat{X}_{P_i} \mid i \in I\}$ are disjoint and do not intersect with $X_{\mathcal{P}} \stackrel{\text{def}}{=} \bigcup_{i \in I} X_{P_i}$ (if this condition does not met, then we replace each SP P_i by an equal to it, in the sense defined at the end of section 4.2, so that this condition will be met). Below we assume that this condition does met, even if private variables of SPs P_i and $P_{i'}$ from $\mathcal{P}$, where $i \neq i'$, have the same designations.

For each DP $\mathcal{P}$ $\mathcal{X}_{\mathcal{P}}$ denotes the set of all variables occurred in $\mathcal{P}$.

A **state** of DP $\mathcal{P}$ is a pair $s = (\theta^s, \{c^s \mid c \in \mathcal{D}_{\mathbf{C}}\})$, where $\theta^s \in \Theta(\mathcal{X}_{\mathcal{P}}), c^s \in \mathcal{D}^*$. A set of all states of a DP $\mathcal{P}$ is denoted by $\Sigma_{\mathcal{P}}$.

$\forall\, s \in \Sigma_{\mathcal{P}}$, $\forall\, e \in \mathcal{E}(\mathcal{X}_{\mathcal{P}})$ the value e^{θ^s} is denoted by e^s.

Let $\mathcal{P} = \{P_i \mid i \in I\}$ be a DP. $\forall\, s \in \Sigma_{\mathcal{P}}, \forall\, i \in I$ $s_i \stackrel{\text{def}}{=} (\theta_i^s, \{c^s \mid c \in \mathcal{D}_{\mathbf{C}}\}) \in \Sigma_{P_i}$, where $\theta_i^s \in \Theta(\mathcal{X}_{P_i})$, $\forall\, x \in \mathcal{X}_{P_i}$ $x^{\theta_i^s} = x^{\theta^s}$.

A state $s \in \Sigma_{\mathcal{P}}$ is said to be **initial** (and is denoted by $0_{\mathcal{P}}$), if $\forall\, i \in I$ $s_i = 0_{P_i}$, **terminal**, if $\forall\, i \in I$ s_i is terminal, and **deadlock** if it is nonterminal, and $\forall\, i \in I$ there is no non-empty transition of P_i from s_i.

Let $\mathcal{P} = \{P_i \mid i \in I\}$ be a DP. **A transition** in $\mathcal{P}$ corresponding to an action $\alpha \in \mathcal{A}_{P_i}$ is a pair (s, s') of states from $\Sigma_{\mathcal{P}}$, such what

$$s_i \stackrel{\alpha}{\to} s_i',\ \forall\, i' \in I \setminus \{i\}\ \ s_{i'} \to s_{i'}'. \tag{6}$$

Property (6) is denoted by $P_i^{v \to v'} : s \to s'$, where $v = at_{P_i}^s$, $v' = at_{P_i}^{s'}$. If (s, s') is a transition of $\mathcal{P}$, then we will denote it by $s \to s'$.

The relationship between states $s, s' \in \Sigma_{\mathcal{P}}$ satisfying (6) can be interpreted as follows: if $\mathcal{P}$ is in the state s at the current time, and from that moment on, a SP $P_i \in \mathcal{P}$ sequentially performed EAs from α, and $\forall\, i' \in I \setminus \{i\}$ $P_{i'}$ did not perform any actions during all this time, then after completion the execution of EAs from α, the new state of $\mathcal{P}$ is s'.

The set $\Sigma_{\mathcal{P}}$ can be considered as a graph in which there is an edge from s to s' labeled α_{P_i} if and only if (6) is true.

An **execution** of a DP $\mathcal{P}$ is a sequence of states $s_0, s_1, \ldots$, such that $s_0 = 0_{\mathcal{P}}$, and each pair s_i, s_{i+1} of neighboring states in this sequence is a transition of $\mathcal{P}$.

A state s of a DP $\mathcal{P}$ is said to be **reachable** if there is a path from $0_{\mathcal{P}}$ to s. Below $\Sigma_{\mathcal{P}}$ denotes the set of reachable states of $\mathcal{P}$.

4.4 A method for constructing a distributed process which is a model of a MPI program

We will consider only such MPI programs that contain

- operators of assignment, conditional transition, loop,
- functions of sending and receiving messages, defined in section 2.2, and
- service MPI functions (`MPI_Init`, etc.) mentioned in the program from section 3.2.

Let Π be a MPI program, $X_\Pi \subseteq \mathcal{X}$ be a set of input variables of Π, and $\varphi_\Pi \in \mathcal{B}$ be an initial condition of Π. A **model** of Π is the DP $\mathcal{P}_\Pi = \{P_i \mid i \geq 0\}$, where $\forall\, i \geq 0$ SP P_i is constructed as follows.

- $\mathcal{X}_{P_i}$ consists of variables from X_{Π}, and i-th copies of private variables of Π, $X_{P_i} = X_{\Pi}$, $\varphi_{P_i} = \varphi_{\Pi}$.
- The graph P_i is constructed by
  - replacing in Π second argument of the function `MPI_Comm_rank` (in the MPI program presented in section 3.2 this is variable `rank`) by i,
  - deleting non-executable parts of the resulting program, and
  - transformation the resulting program into graph form, similarly to how the program in operator form is transformed to a flowchart (with the difference that in flowcharts actions are associated with nodes, and in our model actions are associated with edges).

Message passing functions are represented in P_i by the following actions:

- function (1) is represented by the action $c_r!(e, i, l)$, where
  - e is a term whose value must be equal to the content of the memory segment sent by this function,
  - r and l are the corresponding arguments of function (1) (receiver number and tag, respectively),
- function (2) is represented in P_i by the action $c_i?(e, s, l)$, where
  - e is a term whose value after performing this action must be equal to the received message,
  - s and l are new variables,

  and if the last argument of function (2) has the name q, then the expressions in P_i of the form q`.MPI_SOURCE` and q`.MPI_TAG` are replaced with s and l, respectively,
- representation of function (3) depends on i:
  - for $i = 0$ the function (3) is represented by the action $\circ!e$, where e is a term whose value must be equal to the content of the memory segment sent by this function,
  - for $i \neq 0$ the function (3) is represented by the action $\circ?e$, where e is a term whose value after performing this action must be equal to the received message.

The reduction operation described in section 4.2 can be applied to the constructed graph P_i.

To facilitate an analysis of DP $\mathcal{P}_{\Pi}$, one can add to actions of this DP assignments of the form $\iota := e$, where ι is a new variable (called an **auxiliary** variable), and $e \in \mathcal{E}(\mathcal{X}_{\mathcal{P}_{\Pi}} \sqcup \mathcal{I})$, where $\mathcal{I}$ is a set of auxiliary variables. The assignments of the above form $\iota := e$ are not actually performed actions, they are intended only to express dependencies between values of variables during an execution of the DP. SPs from SPs $P_i \in \mathcal{P}_{\Pi}$ by adding assignments of the above form $\iota := e$, where ι is an auxiliary variable, will be called **augmented** SPs.

5 Modeling and verification of the matrix multiplication MPI program

In this section, we apply the above concepts to modeling and verification of the matrix multiplication MPI program Π described in section 3.

5.1 A model of the matrix multiplication MPI program

Define DP $\mathcal{P}_{\Pi} = \{P_i \mid i \geq 0\}$, which is a model of Π.

We assume that the following variables belong to X_{Π}: A, B (matrix factors), N (number of rows in A), `nprocs`.

$\forall\, i \geq 0$, when constructing SP P_i, the following simplifications are used:

- the function for input of matrix factors `input (a, b)` in line 15 is executed once at the initial moment of execution of the root process, therefore in SP P_0 it is possible to omit actions corresponding to this function, assuming that X_{P_0} has variables A and B, values of which are equal to matrix factors,
- BMP functions in P_0 in lines 16-17 and in P_i ($\forall\, i \geq 1$) in lines 44-45 are executed once, at the initial moment of execution of these SPs, so you can replace the corresponding actions of the form $\circ!e$ and $\circ?e$ on the assumption that B belongs to X_{P_i} ($\forall\, i \geq 1$).

To shorten the notation in SPs below, instead of input variable `nprocs`, we use input variable n, a value of which is equal to `nprocs` $- 1$.

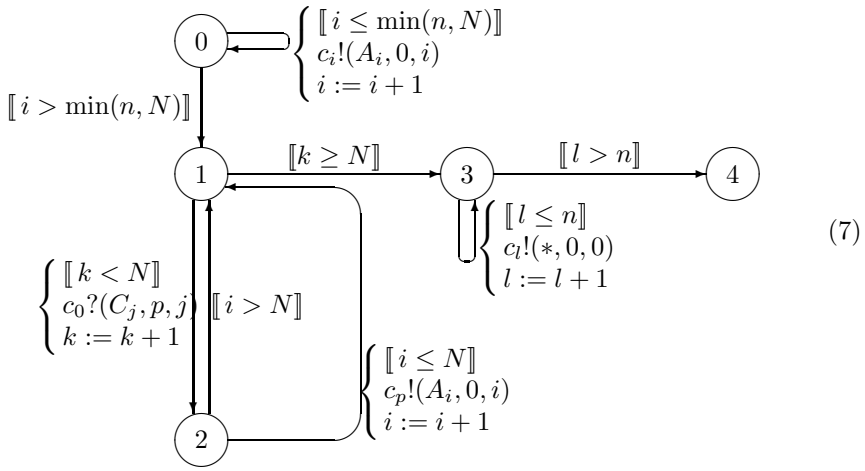
To build SP P_0, only a part of Π is used, located in lines 12-39.

In SP P_0 A and C are names of arrays, components of which are rows of corresponding matrices and are indexed by numbers $1, \ldots, N$, $\forall\, i = 1, \ldots, N$ A_i and C_i denote i-th components of these arrays.

P_0 has the following variables: $X_{P_0} = \{A, B, N, n\}$, $\hat{X}_{P_0} = \{C, i, j, k, l, p\}$.

Initial condition: $\varphi_{P_0} = (N \geq 1) \wedge (i = 1) \wedge (k = 0) \wedge (l = 1)$.

SP P_0 has the form



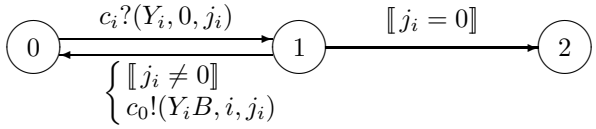
(7)

In SP (7)

- edge $P_0^{0\to 0}$ corresponds to the loop in lines 18-22 of Π, the variable i in the label of this edge corresponds to the expression `count` $+ 1$ in Π,
- the message sent by the `MPI_Send` function in lines 21-22 of Π, is represented by the triple $(A_i, 0, i)$, third component of which is a tag of this message (i.e. a number of the corresponding row in the matrix A),

– edge $P_0^{0\to1}$ corresponds to the exit from this cycle,
– edges $P_0^{1\to2}$ and $P_0^{2\to1}$ correspond to a loop in lines 23-34 of Π,
– variable k corresponds to variable `i` in this loop,
– the function `MPI_Recv` in lines 24-26 and the loop in lines 27-28 of Π are replaced with a single action: receiving a message and writing it to the corresponding row of matrix C,
– edge $P_0^{1\to3}$ corresponds to the exit from this loop,
– edges $P_0^{3\to3}$ and $P_0^{3\to4}$ correspond to the cycle in lines 35-37,
– symbol $*$ in the label of edge $P_0^{3\to3}$ represents the empty string.

To construct SP P_i ($i \geq 1$), only the part of Π located in lines 42-55 is used. P_i has the following variables: $X_{P_i} = \{B\}, \hat{X}_{P_i} = \{Y_i, j_i\}$, where a value of B is the second factor matrix, and values of Y_i are strings of real numbers. P_i has the following form:



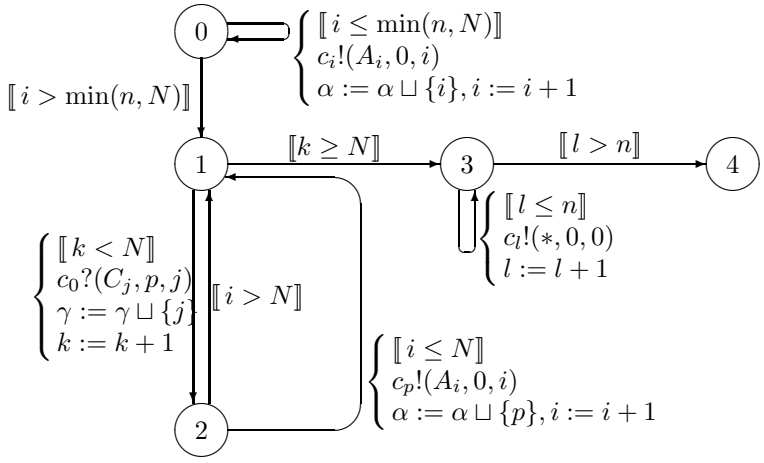
(8)

5.2 Verification of a distributed process that is a model of the matrix multiplication MPI program

To prove the statement that DP $\mathcal{P}_\Pi$ defined in section 5.1, satisfies specification (4), we introduce auxiliary variables α, β, γ, and actions related to these variables that do not affect an execution of the DP. Values of the auxiliary variables have the following meaning: their initial values are $\emptyset$, and $\forall\, s \in \Sigma_{\mathcal{P}_\Pi}$

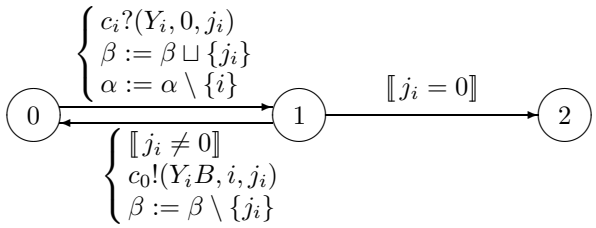
– $\alpha^s \subseteq \{1, \ldots, n\}$, α^s consists of channel numbers from $\{c_1, \ldots, c_n\}$ with non-empty content in state s,
– $\beta^s \subseteq \{1, \ldots, N\}$, β^s consists of numbers of rows of A, for which their product by B is calculated in state s,
– $\gamma^s \subseteq \{1, \ldots, N\}$, γ^s is the set of numbers of rows that P_0 wrote in C during an execution of $\mathcal{P}_\Pi$ up to state s.

Augmented SP P_0 has the form



(9)

$\forall\, i = 1, \ldots, n$ augmented SP P_i has the form



(10)

The use of $\sqcup$ in (9) and (10) for set union operations expresses the statement (following from the following theorem 1) that whenever these operations are performed, their arguments are actually disjoint sets.

Theorem 1.
$\forall\, s \in \Sigma_{\mathcal{P}_\Pi}$, if $at^s_{P_0} \neq 3, 4$, then the following statements are true:

1. $\alpha^s \subseteq \{1, \ldots, i^s - 1\}$,
2. $i^s - 1 \leq N$
3. $|\gamma^s| = k^s \leq N$,
4. if $at^s_{P_0} = 1$ and $k^s < N$, then $k^s < i^s - 1$,
5. $[c_0]^s_2 \cap \alpha^s = \emptyset$,
6. $\forall\, i = 1, \ldots, n$
   (a) $|[c_i]^s| = 1$, if $i \in \alpha^s$, and $|[c_i]^s| = 0$, otherwise,
   (b) $at^s_{P_i} = 1 \;\Rightarrow\; (i \notin \alpha^s) \wedge (j^s_i \notin \gamma^s)$
7. $at^s_{P_0} = 2 \;\Rightarrow\; p^s \notin \alpha^s$, $p^s \in \{1, \ldots, i^s - 1\}$,
8. $[c_1]^s_3 \sqcup \ldots \sqcup [c_n]^s_3 \sqcup \beta^s \sqcup [c_0]^s_3 \sqcup \gamma^s = \{1, \ldots, i^s - 1\}$,
9. $\forall\, p \in \alpha^s$ $[c_p]^s$ has the form $\{(A_i, 0, i)\}$, where $i \in \{1, \ldots, N\}$,
10. each element of $[c_0]^s$ has the form $(A_i B, p, i)$, where $i \in \{1, \ldots, N\}$,
11. $\forall\, j \in \gamma^s$ $C_j = A_j B$,
12. $k^s = N \;\Rightarrow\; \gamma^s = \{1, \ldots, N\}$.

Proof.
All the statements are substantiated inductively: they are true in $0_{\mathcal{P}_\Pi}$, and retain their truth after each transition $s \to s'$ of $\mathcal{P}_\Pi$, where $at^{s'}_{P_0} \neq 3, 4$. ■

Theorem 2.
There are no deadlocks in $\Sigma_{\mathcal{P}_\Pi}$.

Proof.
Let there is a deadlock $s \in \Sigma_{\mathcal{P}_\Pi}$. It is not hard to prove that $at^s_{P_0} \notin \{0, 2, 3\}$, and $\forall\, i = 1, \ldots, n$ $at^s_{P_i} \neq 1$. So, $at^s_{P_0} \in \{1, 4\}$.

1. Let $at^s_{P_0} = 1$. Since s is a deadlock, then $k^s < N$, whence $[c_0]^s = \emptyset$.
   From statement 4 of theorem 1 it follows that $k^s < i^s - 1$, whence, based on statement 8 of theorem 1 and the equality $[c_0]^s = \emptyset$ we get:

$$[c_1]^s_3 \sqcup \ldots \sqcup [c_n]^s_3 \sqcup \beta^s \neq \emptyset. \tag{11}$$

If $\exists\, i \in \{1,\ldots,n\}$: $[c_i]^s \neq \emptyset$, then the assumption that s is a deadlock implies that $at^s_{P_i} \neq 0$, therefore $at^s_{P_i} = 2$, whence it is easy to get that $at^s_{P_0} = 4$. This is possible only if $k^s \geq N$, which contradicts the inequality $k^s < N$. Therefore, $\forall\, i \in \{1,\ldots,n\}$ $[c_i]^s = \emptyset$, and from (11) it follows that $\beta \neq \emptyset$. By analyzing SP P_i $(i = 0,\ldots,n)$ it is easy to prove that this is possible only if $\exists\, i \in \{1,\ldots,n\}$: $at^s_{P_i} = 2$. As stated above, this is impossible.

2. Let $at^s_{P_0} = 4$. Then $k^s \geq N$, whence $|\gamma^s| = k^s = N$. Let s' be first state on a path π from $0_{\mathcal{P}_{\Pi}}$ to s, such that $k^{s'} = N$. It is easy to see that $at^{s'}_{P_0} = 2$. From statements 2 and 8 of theorem 1 it follows that $i^{s'} - 1 = N$, and

$$[c_0]^{s'} = [c_1]^{s'} = \ldots = [c_n]^{s'} = \beta^{s'} = \emptyset,\ \ at^{s'}_{P_1} = 0,\ldots, at^{s'}_{P_n} = 0.$$

The only transition from s' corresponds to the action $[\![i > N]\!]$, and this transition has the form $P_0^{2\to1} : s' \to s''$. The only transition from s'' corresponds to the action $[\![k \geq N]\!]$ and this transition has the form $P_0^{1\to3} : s'' \to s'''$. It is easy to see that all states in the tail of π starting from s''' are not deadlocks.

In both cases, we get a contradiction, which is a consequence of the assumption that $\Sigma_{\mathcal{P}_{\Pi}}$ has a deadlock state. Thus, there are no deadlock states in $\Sigma_{\mathcal{P}_{\Pi}}$. ■

Theorem 3.
Any execution of $\mathcal{P}_{\Pi}$ terminates after a finite sequence of steps.

Proof.
Let there is an infinite execution π of $\mathcal{P}_{\Pi}$. Prove that a number of transitions in π corresponding to actions of P_0 is finite. If this is not so, then

- there are no states $s \in \pi$ such that $at^s_{P_0} = 3$,
- there is a state $s_1 \in \pi$, such that $at^{s_1}_{P_0} = 1$,
- each transition in π, starting from s_1, corresponding to some action P_0, either has the form $P_0^{1\to2} : s \to s'$, or has the form $P_0^{2\to1} : s \to s'$, and the number of transitions of the form $P_0^{1\to2} : s \to s'$ is infinite. This is impossible due to the fact that each such transition increases the value of the variable k, which, according to statement 3 of theorem 1, is bounded by N.

Let $s' \in \pi$ be a state starting from which π does not contain transitions corresponding to actions of P_0, and π' is a tail of π, starting with s'. It is not hard to see that

$$\begin{array}{l}\exists\, i \in \{1,\ldots,n\}\text{: } \pi' \text{ contains infinitely many}\\ \text{transitions of the form } P_i^{0\to1} : s \to s'.\end{array} \tag{12}$$

Because π' does not contain transitions corresponding to actions of P_0, then the value of $|[c_i]^s|$ cannot increase, and according to (12) it decreases infinitely, which is impossible. ■

It follows from the above theorems that each execution of DP $\mathcal{P}_{\Pi}$ is finite and terminates in some terminal state s. From $at^s_{P_0} = 4$ it follows that $|\gamma^s| = k^s = N$, whence, according to statements 11 and 12 of theorem 1, it follows that DP $\mathcal{P}_{\Pi}$ satisfies the specification stated in section3.4. ■

6 Conclusion

In this article, we introduced a new mathematical model of parallel programs, proposed an approach for verifying such programs, and considered an example of verifying a matrix multiplication program based on the proposed model. The main advantage of this approach is the possibility of its application for programs that generate a unlimited set of SPs.

Problems for further research related to the proposed model can be the following: 1). to expand the proposed model by introducing concepts for modeling synchronous message passing, and other mechanisms for organizing parallel execution, 2). to introduce specification language of properties of DPs, in which properties of parallel programs are expressed in terms of observational equivalence, and to elaborate algorithms for recognizing observational equivalence.

References

1. Hugo A. Lopez, E. R. B. Marques, F. Martins, N. Ng, C. Santos, V.T. Vasconcelos, N. Yoshida, Protocol-based verification of message-passing parallel programs, Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015, p. 280-298.
2. S. F. Siegel and G. Gopalakrishnan. Formal Analysis of Message Passing. In R. Jhala and D. Schmidt, editors, Verification, Model Checking, and Abstract Interpretation: 12th International Conference, VMCAI 2011, volume 6538 of Lecture Notes in Computer Science, pages 2-18, (2011).
3. S. F. Siegel, Model Checking Nonblocking MPI Programs. In International Workshop on Verification, Model Checking, and Abstract Interpretation VMCAI 2007, pp 44-58.
4. Siegel, S., Mironova, A., Avrunin, G., Clarke, L.: Combining symbolic execution with model checking to verify parallel numerical programs. ACM Transactions on Software Engineering and Methodology, Volume 17, Issue 2, 2008, 1–34.
5. Sarvani S. Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic Verification of MPI Programs with Reductions in Presence of Split Operations and Relaxed Orderings. In International Conference on Computer Aided Verification CAV 2008, pp 66-79.
6. Gopalakrishnan, G., Kirby, R.M., Siegel, S., Thakur, R., Gropp, W., Lusk, E., De Supinski, B.R., Schulz, M., Bronevetsky, G.: Formal analysis of MPI-based parallel programs. Communications ACM 54(12), 82-91, (2011).
7. V. Forejt, S. Joshi, D. Kroening, G. Narayanaswamy, S. Sharma, Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs, In: ACM Transactions on Programming Languages and Systems, V. 39, Issue 4, 2017, 1–27.
8. Z. Luo, M. Zheng, and S. F. Siegel. Verification of MPI programs using CIVL. In EuroMPI. 6:1-6:11, 2017.
9. W. Hong, Z. Chen, H. Yu, and J. Wang. Evaluation of model checkers by verifying message passing programs. Science China Information Sciences, volume 62, Article number: 200101 (2019).
10. H. Yu, Z. Chen, X. Fu, J. Wang, Z. Su, J. Sun, C. Huang, and W. Dong. Symbolic Verification of Message Passing Interface Programs. In ICSE '20: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, p. 1248-1260, 2020.