COMPUTER SCIENCE | RESEARCH ARTICLE

Petri net modeling and simulation of pipelined redistributions for a deadlock-free system

Stavros I. Souravlas and Manos Roumeliotis

Abstract: The growing use of multiprocessing systems has given rise to the necessity for modeling, verifying, and evaluating their performance in order to fully exploit hardware. The Petri net (PN) formalism is a suitable tool for modeling parallel systems due to its basic characteristics, such as parallelism and synchronization. In addition, the PN formalism allows the incorporation of more details of the real system into the model. Examples of such details include contention for shared resources (like memory) or identification of blocked processes (a definition for blocked processes is found in the Introduction section). In this paper, PNs are considered as a modeling framework to verify and study the performance of parallel pipelined communications. The main strength of the pipelines is that if organized in a proper way, they lead to overlapping of computation, communication, and read/write costs that incur in parallel communications. Most of the well-known pipelined schemes have been evaluated by theoretical analysis, queueing networks, and simulations. Usually, the factors taken into account are scheduling, message classification, and buffer spacing. To the best of our knowledge, there is no work in the literature that uses PN as a modeling tool for verification of the pipeline-based scheme. Apart from verification, a more accurate and complete model should also consider other factors, such as contentions and blocked processes. These factors have a high impact on the performance of a parallel system. The PN model presented in this paper accurately captures the behavior of a pipeline-based parallel communication system. The model considers message scheduling and message classification, while it is deadlock and contention free. Because it is symmetric, it can easily be used for larger systems only with minor changes.

ABOUT THE AUTHORS

Stavros Souravlas and Manos Roumeliotis are faculty members of the Department of Applied Informatics at the University of Macedonia. They are the authors of three books in the field of Digital Logic Design, Digital Systems Modeling and Simulation with VHDL, and Simulation Techniques. Also, Stavros Souravlas Manos Roumeliotis are members of the Computer and Network Systems Technologies (CNST) group at the University of Macedonia. This group conducts research on the fields of digital systems design, parallel and distributed processing, communication networks, hardware description languages, and modeling and simulation of computer and network systems. This work is a part of a general research project ran by the CNST group on the possible faults that incur when applying parallel algorithms on a multiprocessor network.

PUBLIC INTEREST STATEMENT

This paper presents a Petri net-based model used to verify and evaluate the performance of pipelined parallel distributions. It precisely captures the behavior of a pipeline-based parallel communication system. The model considers message scheduling and message classification, while it is deadlock and contention free. Because it is symmetric, it can easily be used for larger systems only with minor changes.

The model is composed of two subnetworks: the reading subnetwork and the pipeline execution subnetwork. The reading subnetwork implements the task of preparing the necessary pipeline tasks. The pipeline execution subnetwork implements the actual communication by executing the series of tasks. The model (and thus the real system) maintains its safety by preventing a pipeline from starting until all pipeline tasks have been formed. Also, it avoids deadlocks by preventing a new communication to start before all the other segments have been assigned the proper tasks.
captures the behavior of the pipeline-based parallel communication system. The model considers synchronization, message scheduling, and message classification, while it is proven to be free of deadlocks and contentions. Also, the model is characterized by symmetry, so it can be used for large and complex systems.

**Subjects:** Systems & Computer Architecture; Design; Systems Architecture

**Keywords:** Petri nets; parallel systems; pipeline; performance evaluation; block cyclic redistributions

1. **Introduction-related work**

The problem of data distribution between several processors is very important, affecting the efficiency of parallel algorithms. As a parallel program is being executed, it may require a different distribution of data between the processors or redistribution.

The principal issues that need to be considered to model a pipeline-based communication are (1) **total redistribution cost**, (2) **message scheduling**, (3) **message classification**, (4) **load balancing**, (5) **contentions**, and (6) **blocked processes**.

(a) **Total redistribution cost**: It is the total cost to redistribute data between several processors. It is composed of the **index computation overhead**, the **total communication overhead**, and the **R/W (or I/O) overhead**. The index computation overhead refers to computing the target processor and the memory positions where each data element will be located. The total communication overhead incurs during data distribution between parallel processors. The R/W overhead refers to the time consumed by R/W operations.

(b) **Message scheduling**: The term refers to the organization of message exchanges into structured communication steps, such that the total redistribution cost is minimized.

(c) **Message classification**: The term refers to dividing the data to be redistributed into homogeneous (in terms of communication cost) groups to effectively organize their redistribution.

(d) **Load balancing**: The ability of having all communication links equally or nearly equally loaded to avoid network congestions.

(e) **Contention**: This refers to conflicting transferring processes that distribute data to the same target processor at the same time.

(f) **Blocked processes**: A process is blocked if it waits for the execution of another process that never occurs.

Several pipeline-based techniques have been proposed in the literature (King, Chou, & Ni, 1990; Preud’homme, Sopena, Thomas, & Folliot, 2012; Rodrigues, Wheeler, & Kogge, 2008). The pipeline techniques have been widely studied (Caron & Desprez, 2005; Jayachandran & Abdelzaher, 2007; Kashif, Gholamian, Pelizzoni, Patel, & Fischmeister, 2013; Kuntraruk, Pottenger, & Ross, 2005; Souravlas & Roumeliotis, 2004, 2014 among others) and used to schedule interprocessor communication. In Caron and Desprez (2005), pipelining is combined with out-of-core techniques (OOC) to overlap the computation, communication, and R/W overheads. Scheduling is based on dividing memory into three memory blocks and then loading three blocks of data into memory. These blocks are three times smaller than what should be loaded without this segmentation. The communication costs overlap because during each period, the first memory block performs I/O, computation is performed on the data stored in the second block, and data from the third block is distributed to a nearby processor. No actual message classification incurs. Data are just fragmented into three pieces and transferred. Based on scheduling, no contentions should occur. During anytime, a memory block clearly receives one data block. The communication model does not take into account the issue of blocked processes. Assuming that there is a proper synchronization of the pipeline segments (so that no process can block the execution of another process), the communication tasks pass from
one segment to another and delays are introduced on the writing processes to avoid idle times. A similar approach is presented in Jayachandran and Abdelzaher (2007), but it imposes a restriction on the usage of the pipelines: every task should finish before proceeding to the next pipeline segment.

In Kashif et al. (2013), the worst-case response times of real-time applications on multiprocessor systems are computed. The proposed technique schedules a simple pipelined communication operation for data distribution. The model consists of a set of processing resources interconnected with pipelined communication resources (CRs). Data transmitted on the CRs travel through the first segment followed by the next, and so on. Simultaneous transmission of data on segments is allowed. This means that while data are being transmitted on a later segment, new data can be transmitted on an earlier segment in parallel. However, if this situation is not handled carefully, it can lead to blocked processes. Pipeline segments should be carefully released first, before handling new transmissions.

The work in Kuntraruk et al. (2005) addresses the problem of developing a resource estimation model for applications executed within a parallel pipeline model of execution. The model estimates the computation and communication complexities for parallel pipelined applications. It includes two components: the ones that execute pipelined application tasks and the ones that perform merge operations. In the beginning, there are $P$ tasks processed on $P$ processors, one task per processor. Afterwards, at every step, the tasks are reduced in half since half of the processors are merging data received from the previous step. The model pays no attention to conflicts that could easily arise when different data volumes are carried over the network of processors. Also, there is no concern about possible blocked processes.

An interesting approach for modeling pipeline parallelism is given in Navarro et al. (2009). The authors develop a series of analytical models based on queueing theory for several parallel pipeline templates, which are modeled as closed or open queueing systems. Specifically, each pipeline segment is treated as a $M/M/c_i/N/K$ queue (for closed systems) or as a $M/M/c_i$ queue (for open systems). The models assure load balancing over the network. Since the proposed models are based on queues, contention is not avoided between messages that try to enter the queue. Things deteriorate if there is not much space in the queues. Necessarily, these models (unlike most of the models described) have to take into account the limited memory (buffer space). Simulations on real systems were used to verify that the queuing pipeline models capture the behavior of parallel systems faithfully. A related approach in Ties et al. (2007) marks the pipeline segments and tries to track the data communications performed in each of them. Also, an open system approach model is proposed by Liao et al. (2004), with the same factors taken into account. The latest approaches mentioned do not guarantee load balancing between processor sets.

Many other pipeline-based parallel communication models have been presented in the literature (King et al., 1990; Preud’homme et al., 2012; Rodrigues et al., 2008; Zhang & Deng, 2002). Generally, the models presented are basically concerned with maintaining some load balancing on the network during the pipelined distributions, with little or no attention paid on the problem of contentsions and blocked processes. All these models also use simulation as the verification and performance study tool. Table 1 summarizes the factors addressed by the pipeline communication models discussed in this section. Note that all of the models involve some communication scheduling and load balancing (the messages distributed are of the same size) and none of them considers blocked processes. Also, most of the models assume that buffer space is enough to handle the distributed data and do not include a straightforward contention-preventing mechanism.

This paper introduces a Petri net (PN) model for modeling and simulating pipelined and deadlock-free parallel communications. PNs are used to examine the sequence of executed tasks (Granda, Drake, & Gregorio, 1992). The study of the process sequence is very important to avoid faults such as deadlocks [(some very general ideas about modeling pipelined parallel communication with PN can be found in Zhao, Liu, Dou, and Yang (2012)]. A deadlock-free scheme enhances the performance of
any communication system. Generally, deadlocks occur when processes stay blocked for ever (waiting for an event caused by another process that never occurs) and in such cases, probably the whole system needs to be restarted. A block cyclic redistribution scheme can suffer from deadlock situations since each target block is formed during runtime and after a series of interrelated processes, which are described in Section 4.

The rest of the paper is organized as follows: Section 2 presents the background of the model, which is based on block cyclic(r) to block cyclic(s) distributions. Section 3 describes the pipelined communication, which is modeled via PN in Section 4. Section 5 presents some simulations for the complete PN model of Section 5, for three different communication scenarios. Section 6 concludes this paper.

2. Background
The model presented in this work has it is mathematical background on the well-known block cyclic redistribution problem, so this section briefly introduces the definitions required.

**Definition 1** Data array is an array of size $M$ used to represent the redistributed data. An array element is an element of the redistributed data indexed with $i$. Indexing begins from zero, thus, $i \in [0 \ldots M - 1]$.

**Definition 2** A processor grid can be represented by a two-dimensional (2D) table called communication grid $\Pi = \{(p, q) \in [0 \ldots P - 1] \times [0 \ldots Q - 1]\}$. Obviously, $p$ is the source processor index, $q$ is the destination processor index, while $P, Q$ represent the total number of sending and receiving processors, respectively.

**Definition 3** Data distributed in a block cyclic fashion are divided into data blocks. If each data block has $r$ elements, then, provided that $M$ divides $r$, the data array will be divided into $M_b$ blocks where: $M_b = \frac{M}{r}$. If $M$ does not divide $r$, then $M_b = \frac{M}{r} + 1$. We use variable $l$ as a block index that relates data blocks to the processors of the communication grid in a cyclic manner. Therefore, $l$ lies in $[0 \ldots \frac{M_b}{P}]$.
Table 2. Definitions of variables in this paper

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Number of sending nodes</td>
<td>s</td>
<td>Target distribution block size</td>
</tr>
<tr>
<td>Q</td>
<td>Number of receiving nodes</td>
<td>i</td>
<td>Source distribution block index</td>
</tr>
<tr>
<td>M</td>
<td>Size of data array</td>
<td>m</td>
<td>Target distribution block index</td>
</tr>
<tr>
<td>R</td>
<td>Source distribution</td>
<td>x</td>
<td>Source distribution element position</td>
</tr>
<tr>
<td>R'</td>
<td>Target distribution</td>
<td>y</td>
<td>Target distribution element position</td>
</tr>
<tr>
<td>p</td>
<td>Source processor index</td>
<td>g</td>
<td>Greatest common divisor of Pr, Qs</td>
</tr>
<tr>
<td>q</td>
<td>Destination processor index</td>
<td>L</td>
<td>Least common multiplier of Pr, Qs</td>
</tr>
<tr>
<td>r</td>
<td>Source distribution block size</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. Pipelined communication

This section presents the pipelined interprocessor communication. Each pipeline includes a number of tasks responsible for the communication between carefully selected processor pairs. The main properties of the pipeline operations and their tasks are:

1. Each pipeline task handles the transmission of data between processor pairs that have the same communication cost.
2. A pipeline operation cannot include more than one task that handles message transmissions of a cost.
3. The time required for the execution of a task equals the communication cost of the processor pairs it includes.
4. The time required for the execution of a pipeline operation equals the execution time of its longest task.
5. All tasks are scheduled in such a way that receiving processors get one message at a time, thus congestions on the receiving ports are avoided.
6. The pipeline will include a number of segments (the role of segments in the communication will be explained in Section 3.3) equal to the number of different costs that exist in the scheme.
7. The time the processors remain idle is minimized.

The pipelined data distribution is composed of three stages: (1) generating the pipeline tasks, (2) reading messages from memory, and (3) transferring the messages and writing them to the target processors’ memory. In the next sections, details for each stage are presented.

3.1. Stage 1: Generating the pipeline tasks

The pipeline tasks must be scheduled in such a way that receiving processors get one message at a time. To satisfy this requirement, each task must include a number of distributions of same cost to different destination processors. Therefore, classes are used to group all the communicating processor pairs with respect to the cost of such communication. A processor pair lies in class \( b(k) \), if \( k = (p_r - q_s) \mod g \). The class processor table (CPT) shows the class of each processor pair and the communication cost of this class. Consider a redistribution with \( P = Q = 9 \), \( r = 4 \), and \( s = 5 \). In this case, \( g = 9 \). The CPT for this redistribution example is shown in Table 3. For example, if \( (p, q) = (4, 3) \), then \( p_r - q_s = 16 - 15 = 1 \). Thus, \( (p_r - q_s) \mod g = 1 \mod 9 = 1 \). This means that the processor pair \((4,3)\) belongs to the class \( k = 1 \). The cost of communication for each class is computed as the number of quadruples \((x, y, l, m)\) that satisfy Equation 1, for a given set \((p, q)\).

Having defined the classes, it remains to: (1) find the number of pipeline operations and the number of their tasks, (2) define an upper bound for the number of processor pairs selected from each class for a pipeline task, and (3) define the number of classes from which the processor pairs are selected to have a minimum of \( Q \) transmissions (one message for each destination processor) in each pipelined communication.

Table 3. CPT for \( P = Q = 9, r = 4 \), and \( s = 5 \)

<table>
<thead>
<tr>
<th>Class</th>
<th>Processor pairs</th>
<th>Communication cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>b(0)</td>
<td>(0, 0), (8, 1), (7, 2), (6, 3), (5, 4), (4, 5), (3, 6), (2, 7), (1, 8)</td>
<td>4</td>
</tr>
<tr>
<td>b(1)</td>
<td>(7, 0), (6, 1), (5, 2), (4, 3), (3, 4), (2, 5), (1, 6), (0, 7), (8, 8)</td>
<td>4</td>
</tr>
<tr>
<td>b(2)</td>
<td>(5, 0), (4, 1), (3, 2), (2, 3), (1, 4), (0, 5), (8, 6), (7, 7), (6, 8)</td>
<td>4</td>
</tr>
<tr>
<td>b(3)</td>
<td>(3, 0), (2, 1), (1, 2), (0, 3), (8, 4), (7, 5), (6, 6), (5, 7), (4, 8)</td>
<td>4</td>
</tr>
<tr>
<td>b(4)</td>
<td>(1, 0), (0, 1), (8, 2), (7, 3), (6, 4), (5, 5), (4, 6), (3, 7), (2, 8)</td>
<td>4</td>
</tr>
<tr>
<td>b(6)</td>
<td>(6, 0), (5, 1), (4, 2), (3, 3), (2, 4), (1, 5), (0, 6), (8, 7), (7, 8)</td>
<td>4</td>
</tr>
<tr>
<td>b(7)</td>
<td>(4, 0), (3, 1), (2, 2), (1, 3), (0, 4), (8, 5), (7, 6), (6, 7), (5, 8)</td>
<td>4</td>
</tr>
<tr>
<td>b(8)</td>
<td>(2, 0), (1, 1), (0, 2), (8, 3), (7, 4), (6, 5), (5, 6), (4, 7), (3, 8)</td>
<td>4</td>
</tr>
</tbody>
</table>
To minimize the time the processors remain idle, each pipeline must be scheduled to have a maximum number of tasks; in other words, to transfer as much data as possible with a single pipeline operation. If the number of different communication costs found in all classes is \( d \), then a pipeline operation has at most \( d \) tasks and can satisfy up to \( dQ \) message transmissions, without contentions.

Theorem 1 (for proof see Desprez et al., 1998a) is used to define an upper bound for the number of processor pairs in class \( b(k) \) that will be added in a pipeline task. Initially, we set \( s' = \gcd(s, P) \) and \( r' = \gcd(r, Q) \). Since \( s' \) divides \( P \) and \( r' \) divides \( Q \), there exist integers \( P' \) and \( Q' \) such that: 

\[
P' = P/s' \quad \text{and} \quad Q' = Q/r'.
\]

Also, we set \( g_0 = \gcd(P', Q') \).

**Theorem 1** Each class includes exactly \( \frac{g_0}{g} \) processor pairs. Theorem 1 leads to the following corollaries:

1. The number of sending requests to a destination inside a class is \( P/g_0 \).
2. There are exactly \( Q' \) different destinations inside each class, thus a pipeline task can satisfy no more than \( Q' \) communications between processor pairs of a class because this would cause contentions.

To define the number of classes from which processor pairs are selected for a minimum of \( Q \) communications for a pipeline operation, Proposition 1 will be used.

**Proposition 1** For a pipeline operation with minimum number of \( Q \) communications, the communicating processor pairs must be selected from \( r' \) different classes.

The minimum number of message exchanges for a pipeline operation corresponds to “one message for each destination processor”, that is, \( Q \) messages in total. A pipeline task can satisfy at most \( Q \) communications from one class, otherwise contentions will occur. From the relationship \( Q = Q' r' \), one can easily conclude that the processor pairs must be selected from \( r' \) classes to complete \( Q \) transmissions. The generation of the pipeline operations and their tasks can be described in a series of well-defined steps as follows:

**Step 1:** Solve (Equation 3) for all processor pairs \( (p,q) \) to define the processor classes and create the CPT.

**Step 2:** Find the total communication cost for each \( b(k) \) by computing the number of quadruples \((l, m, x, y)\) that satisfy Equation 1. Since each class includes messages of the same cost, only one computation is needed for a pair \((p, q)\). All other processor pairs in the same class would have the same cost. Afterwards, define the value of different costs that exist for this distribution, \( d \).

**Step 3:** Start from the class \( b(k) \), for which the communication cost \( C_{b(k)} \) is minimum, and get \( Q' \) processor pairs. If the pairs selected from \( b(k) \) can form a task of \( Q \) transmissions, that is, if \( Q = Q' \), move to Step 4. Otherwise, check if there is a class of the same cost as \( b(k) \) to add up to \( Q - Q' \) pairs. In either case, the processor pairs that task \( I_i \) should include must be such that all destination processor indices differ: \( T_i = (p_{\mu_0}, q_{\mu_0}), (p_{\mu_1}, q_{\mu_1}), \ldots (p_{\mu_j}, q_{\mu_j}) \), where \( \mu_0 \neq \mu_1 \neq \ldots \neq \mu_{Q'} \).

**Step 4:** Find the class \( b(k) \) with the next communication cost and repeat Step 3. Tasks with the same communication cost are not allowed in the same pipeline operation. Once a pipeline includes \( dQ \) message exchanges, it is completed. Go to Step 5.

**Step 5:** Check the value of \( d \) to find the number of different costs for the rest of the processor pairs and use Steps 3 and 4 to create the next pipeline operation.

**Step 6:** When all processor pairs are added in a pipeline operation, terminate, if not, return to Step 1.

Consider the redistribution for \( P = Q = 9, r = 4, \) and \( s = 5 \). In this case, \( g = 9 \). The CPT is shown in...
Table 3. Also (see the last column of Table 3), \( d = 4 \) since there are four different communication costs in the scheme varying from 1 to 4 time units. According to Step 3, we get \( Q' = 9 \) processor pairs from \( r' = 1 \) class to create a task of \( Q = 9 \) transmissions. We can select pairs from any of the two classes \( b(4) \) and \( b(6) \) since they have the same communication cost of one time unit. Suppose that we select from class \( b(4) \). These processor pairs will form the first task \( T_0 \) of the first pipeline operation.

According to Step 4, the processor pairs of class \( b(6) \) cannot be used in any of the tasks for this pipeline because the same communication cost of one unit will appear twice for all destinations. For the same reason, the classes \( b(2) \) and \( b(8) \), \( b(0) \) and \( b(1) \), and \( b(3) \) and \( b(7) \) are mutually exclusive.

In Step 5, the value of \( d \) is checked to find the number of different costs in the remaining classes \( b(1), b(6), b(7), \) and \( b(8) \). We have \( d = 4 \). Therefore, Steps 3 and 4 are used to create the second pipeline (the two pipeline operations are shown in Table 4).

Once the pipeline operations and their tasks are scheduled, the messages must be read from local processor memories and prepared for distribution. This stage is described in the next section.

### 3.2. Initialization—reading messages from memory

This stage involves computing the local memory positions where the data to be distributed reside. Using the terms of Table 1 and Equation 1, one can describe the reading stage as follows: the reading stage computes the local positions \( x \) of the data elements to be redistributed. These elements reside in block \( l \) of the source processors’ \( (p) \) memory. All this information can be easily obtained when (Equation 1) is solved. As an example, consider the transfer of data blocks towards processor \( q = 0 \) in a redistribution problem with parameters \( P = Q = 9, r = 4, \) and \( s = 5 \). Table 5 gives solutions of equation (Equation 1) when \( q = 0 \) and \( p \in [0, 8] \).

Suppose that we want to know the position of the data elements scheduled to be distributed from source processor \( p = 7 \) to target processor \( q = 0 \). As shown in Table 5, these elements reside in block \( l = 0 \) and their local position inside the block is defined by \( x \), that is, \( 0, 1, 2, \) and \( 3 \). The upper part of Figure 1 shows all the elements that will move to \( q = 0 \) and their initial position in the source processors. These positions are computed from Equation 1, as shown in Table 5. Once the initialization computations are done, the pipelines are ready for execution.

### 3.3. Transferring the messages and writing to the target processors’ memory

When the pipelines execute, they generate a number of communications between several processors. It is important to note that pipeline operations are executed sequentially (one after the other) but their

| Table 4. Pipeline operations and its tasks for \( P = Q = 9, r = 4, \) and \( s = 5 \) |
|------------------------------------------|------------------|---------------------|
| **Pipeline Task** | **Communicating Processor Pairs (p,q)** | **Communication Cost** |
| First pipeline |
| \( T_0 \) | \( \{1, 0, 0, 1, 8, 2, 7, 3, 6, 4, 5, 5, 6, 3, 7, 2, 8\} \) | 1 |
| \( T_1 \) | \( \{3, 0, 2, 1, 1, 2, 0, 3, 8, 4, 7, 5, 6, 6, 5, 7, 4, 8\} \) | 2 |
| \( T_2 \) | \( \{5, 0, 4, 1, 3, 2, 2, 3, 1, 4, 0, 5, 8, 6, 7, 7, 6, 8\} \) | 3 |
| \( T_3 \) | \( \{7, 0, 6, 1, 5, 2, 4, 3, 3, 4, 2, 5, 1, 6, 0, 7, 8, 8\} \) | 4 |
| Second pipeline |
| \( T_0 \) | \( \{6, 0, 5, 1, 4, 2, 3, 3, 2, 4, 1, 5, 0, 6, 8, 7, 7, 8\} \) | 1 |
| \( T_1 \) | \( \{4, 0, 3, 1, 2, 2, 1, 3, 0, 4, 8, 5, 7, 6, 6, 7, 5, 8\} \) | 2 |
| \( T_2 \) | \( \{2, 0, 1, 1, 0, 2, 8, 3, 7, 4, 6, 5, 1, 6, 4, 7, 3, 8\} \) | 3 |
| \( T_3 \) | \( \{0, 0, 8, 1, 7, 2, 6, 3, 5, 4, 4, 5, 3, 6, 2, 7, 1, 8\} \) | 4 |
tasks are parallel. Figure 2 shows the execution of the two pipeline operations for the redistribution with parameters \( P = Q = 9 \), \( r = 4 \), and \( s = 5 \). Each pipeline operation is composed of four tasks \((T_0 - T_3)\).

The horizontal axis displays the time in time units, while the vertical axis gives the pipeline segment. The role of a segment is to handle the distribution process performed by a pipeline task \( T_i \). When a task is handled by the \( n \)th out of \( d \) segments, it is scheduled to be the \( n \)th to complete its distribution job. For example, in Figure 2, there are four segments that handle four transferring tasks. As tasks move “downwards” from segment 4 to segment 1, they are approaching their completion. Apparently, in this figure, \( T_0 \) is to finish first, as it is the “cheapest” task (one time unit, see Table 4). The time required for the execution of a task equals the communication cost of the processor pairs it includes. From the previous discussion in Section 3.1, it is obvious that the pipeline tasks are completed at different times. Since each task cannot contain more than a message to a specific destination, contentions at the receiving processors’ ports are avoided. To make it more clear, consider the four tasks shown in Table 4. All tasks handle messages to the same target nodes; however, congestions are avoided since these tasks complete at different times.
Each of the tasks performs a partial transferring job, that is, it “adds” elements to data blocks at a certain time. Now, let us examine how these tasks are executed during communication to processor $q = 0$. Suppose that communication starts at time $t = 0$. By the end of time $t = 1$, the first task $T_0$ is complete (see Figure 2). This means that one element from source $p = 1$ (note in Table 4 that $p = 1$ sends data to $q = 0$ during execution of $T_0$) is transferred to its new location. By the end of time $t = 2$ two more elements are added from $p = 3$. The first pipeline operation completes at $t = 4$ time units. In the very same manner, the second pipeline operation starts execution at time $t = 5$. At each time unit, a task completes and adds elements to $q = 0$. At $t = 8$, the distribution to $q = 0$ is complete. Similarly, all destination processors receive their data blocks during the same period of time.

When pipeline execution completes, the distributed elements become parts of newly formed blocks indexed by $m$ in the memories of the target processors $q$. Their new local position inside the blocks is defined by $y$. For example, the lower part of Figure 1 shows the newly formed blocks in the memory of $q = 0$. It is clear that each new block has five elements. The block $m = 0$ is formed by four elements transferred from $p = 0$ and one element transferred from $q = 1$. The local position of these five data elements in the newly formed block is given by $y$, that is, 0, 1, 2, 3 (for elements from $p = 0$), and 4 (for the one element from $p = 1$). The lower part of Figure 1 shows the new position of the elements distributed to $q = 0$ in their new blocks $m = 0, 1, 2, 3$. Figure 3 shows the formulation of these blocks over time, during the execution of the pipeline task included in the two pipelines shown in Table 4. Assuming that communication starts at time $t = 1$, at $t = 2$, the first task will be completed. Therefore, the target processor $q = 0$ will have received one data block from the sending processor $p = 1$ (see task $T_0$ of the first pipeline in Table 4). According to the results in Table 5, for $(p, q) = (1, 0)$, this block will be stored in position $y = 4$ of the target block $m = 0$ (recall that block positions indexes start from 0, so this element occupies the last position of the block). Similarly, at $t = 2$, task 2 is completed. Therefore, the target processor $q = 0$ will have received two data blocks from the sending processor $p = 3$ (see task $T_0$ of the first pipeline in Table 4). According to the results in Table 5, for $(p, q) = (3, 0)$, this block will be stored in positions $y = 3$ and $y = 4$ of the target block $m = 1$.

4. The PN pipelined communication model

As described in Section 2, the pipeline based models of communication in the literature do not consider the problem of deadlocks (blocked processes), while they seldom take into account the contents. The goal here is twofold: (1) use the PN model as a tool to verify that the communication model of Section 3 is deadlock and conflict-free and (2) obtain the performance metrics for three different distribution scenarios from the PN model via a discrete event simulations.

The system model under consideration comprises of a number of pipeline tasks in a single pipeline operation that execute as described in Sections 3.1–3.3. Apparently, a distribution problem can
include a variable number of operations with a variable number of tasks. Since all pipeline operations can follow the same pattern (execute a number of parallel tasks), it is important to design a symmetric model, so that it can be applied for variable number of tasks with minor changes. From a modeling perspective, there are three occurrences of interest: (1) generation of the pipeline tasks, (2) execution of the pipelines, and (3) handling the pipeline segments. This section presents a fully symmetric PN model for these occurrences and performs deadlock and safeness analyses to verify that the models (and consequently the pipeline communication schedule) do not suffer from blocked processes or contentions. Notice that occurrences (2) and (3) are closely related, thus presented in the same subsection. Before that some preliminaries regarding PN are required.

4.1. PN preliminaries

This subsection briefly presents the basic PN notations required in this work. A PN is a set of two different types of nodes: places (pictured with circles) and transitions (pictured with bars). Places and transitions are connected via directed arcs from places to transitions and vice versa. If an arc is directed from node $A$ to node $B$, then $A$ is an input to $B$. The state of a PN changes when it is executed. The execution is controlled by the tokens placed inside nodes. When the PN is executed, a number of tokens are removed from their current place and are located in different places. The distribution of tokens in the places defines the state of the net. This distribution is referred as marking of the PN. Apparently, when a system is initialized, it must have an initial state or initial marking. In this paper, every marking (initial or later) is denoted by $\mu_i = (P_i, \ldots, P_{i+k}, \ldots, P_n)$, where $i$ is the marking index ($i = 1$ denotes the initial marking), $n$ is the number of places, and $P_i$ is every place that has a token in this marking.

A change of a PN’s state (that is, the movement of tokens) occurs when the transitions are enabled to fire and this is true when all of its input places have a token. To describe that fact that a transition’s firing changed the marking from $\mu_i$ to $\mu_{i+1}$ the following notation is used: $\mu_i \rightarrow \mu_{i+1}$ or $(P_i, \ldots, P_{i+k}, \ldots, P_n) \rightarrow (P_i, \ldots, P_{i+k}, \ldots, P_n)$, where () indicates the new set of token-holding places. As an example, consider the PN of Figure 4(a). The initial marking is $\mu_1 = (P_3, P_4)$ and the only enabled transition is $t_3$ ($t_2$ is not enabled, although $P_3$ has a token because its input place $P_2$ has no token). When $t_3$ fires, the token will be removed from $P_4$ and two new tokens will be placed, one in $P_3$ and one in $P_4$, resulting in $\mu_2 = (P_2, P_3, P_4)$. 

![Figure 3. Memory writing over time for $P=Q=9, r=4$, and $s=5$.](image-url)
A very important issue about PN is that a sequence of firings can result in a marking $\mu$, where no transition is enabled. This would drive the model in a deadlock (process being blocked by another process). This means that there are a number of unwanted states that can lead to a deadlock. Spotting these states is a very important issue when modeling because it uncovers the sequence of executing events that can lead the model, and consequently the real system, to deadlock. The tool used to analyze the PN model for deadlocks is called reachability tree. A reachability tree is a set of nodes that represents all possible markings of a net (if the set is finite) caused by the firing of transitions. Figure 4(b) shows the reachability tree for the example of Figure 4(b). The transition that causes every new marking is written near the arrows. The parentheses above places show the number of tokens in every place. If no parentheses are included, the number of tokens is 1. In the case of Figure 4(b), no place does store more than one token, but this is not always the case.

Peterson (1997) introduced two basic rules for the reachability analysis: (1) if a newly formed marking is equal to an existing one on the path from the root (which is an initial marking) to this new marking, then this marking is a terminal node. This means that if a new marking is equal to a previous one, then all markings reachable from it are already added to the reachability tree, and (2) if a newly formed marking $y$ is greater than a previous marking $x$, then all possible firings from marking $x$ are also possible from marking $y$. In this case, the components of $y$ which are greater than the corresponding components of $x$ are replaced by the symbol $\omega$, where $\omega$ is a value arbitrarily large compared to a natural number $a$. Also, the sequence of firings that lead from $x$ to $y$ can be repeated endlessly, resulting every time in an increase of the number of tokens in the corresponding positions $P_i$. The symbol $\omega$ is used to denote an arbitrarily large number of tokens in $P_i$, and the notation $P_i^{(n)}$ is used. From Peterson’s rules, we derive Proposition 2 that gives a condition under which a model does not reach a deadlock.

**Proposition 2** A PN model will never reach a deadlock if for all markings that are roots to subtrees formed on the reachability tree, it is possible to find a terminal node or a marking greater than the root.

In a PN model, the places represent conditions, while the transitions represent events. The presence of a token in a place shows that a condition is true. Since a condition is either true or false, there is no point in having more than a token in a place. Thus, especially when modeling hardware, one of the most important characteristic of PN is safety. A place of a PN is safe if its tokens are never more than one. In other words, therefore, safeness is violated when a sequence of firings puts two or more tokens in a place. In a parallel pipelined communication model, safeness ensures that there are no conflicting processes. Two or more processes are in conflict if, at the same time, they try to read the data blocks from the same source processors or to distribute and write data to the same target processors. The reachability analysis can show if multiple tokens are put in one place. Deadlock and safety analyses are provided for the proposed models in the next subsections.
4.2. Modeling the pipeline generation

The background and the steps required to generate the pipelines were presented in Section 3.1. Figure 5(a) presents the PN model and Figure 5(b) shows the pipeline generation in pseudo-code form. The part in the square is the reading subnetwork. Out of the reading subnetwork, there are two places, $P_E$ and $P_G$ and two transitions, $t_E$ and $t_G$, their roles described in the following. The places and transitions of the model are described as follows:

**Places**

- $P_G$: Generation request
- $P_E$: Transfer request (pipeline execution)
- $P_1$: Lowest cost class
- $P_2$: $\bar{Q}$ messages
- $P_3$: While condition
- $P_4$: If condition
- $P_5$: Unread messages from lowest cost class
- $P_6$: End System available to generate new operation

**Transitions**

- $t_G$: Get lowest cost class
- $t_1$: Read $Q - \bar{Q}$ messages
- $t_2$: Check condition $\bar{Q} < Q$
- $t_3$: Assign FALSE to condition
- $t_4$: Assign TRUE to condition
- $t_5$: Terminate reading
- $t_6$: Get to next lowest cost class
- $t_7$: Read unread messages from a same cost class

---

**Generating the Pipeline Tasks**

1. Get $Q$ communicating pairs from a minimum communication cost class.
2. While $\bar{Q} < Q$ do
   
   - Get next class with equal cost
   - Add another $Q$ communications

   // $Q$ communications completed

3. If there is a class not added
   
   - Get it and go to Step 3.

   else end;

   // All classes added to pipelines

---

Figure 5. Pipeline generation model.
The pipeline generation of model of Figure 5 is a repeating process which is initialized by a generation request provided that there is no pipeline generation in process. The initial marking, \( \mu_1 = (P_1) \), indicates that the system is available to generate a new pipeline operation. Once \( P_G \) gets a token, \( t_G \) is enabled to fire and cause a series of firings. Firing of \( t_G \) will cause one token from \( P_G \) and one from \( P_6 \) to be removed and one token to be placed to \( P_1 \) (the lowest cost class is defined). This will produce \( \mu_2 = (P_1) \) and enable \( t_1 \). Once \( t_1 \) fires, a token is removed from \( P_1 \) and one token is be placed to \( P_2 \) (Q messages generated). The new marking is \( \mu_3 = (P_2) \) and \( t_2 \) is enabled. Then, a while condition should be checked to decide if more messages are required from the same class. When \( t_2 \) fires, a token is placed in \( P_3 \) [marking \( \mu_4 = (P_3) \)]. This enables two transitions (\( t_3 \) and \( t_4 \)), but only one can fire. It should be stressed that PN does not have a mechanism to decide which of the enabled transitions will fire. This depends on the designer of the model. When \( t_4 \) is selected, there is a cyclic execution of firings, which repeats until \( Q = Q \) and creates repeatedly the markings \( \mu_5 = (P_2), \mu_6 = (P_2), \) and \( \mu_7 = (P_3) \). When \( t_3 \) is selected, a token is placed to \( P_4 \) (marking \( \mu_8 = (P_4) \)). As with the while condition, the system now has to check if there is a class of a specific cost not added. If so, \( t_6 \) fires to cause \( \mu_9 = (P_6) \). This means that the sequence of firings described will be repeated. If not, \( t_6 \) fires to cause \( \mu_9 = (P_6, P_6) \). This means that the system has terminated the pipeline generation and it is available to generate a new one. Also, a transfer request is activated (pipelines can be executed, as described in Section 4.3).

At this point, an anti-paradigm is necessary to stress the importance of modeling. Assume that the arc from \( P_6 \) to \( t_5 \) is removed. Also, assume that a pipeline generation is in process (say, is checking the while statement, that is, a token is in \( P_6 \)). If a second generation request is made, the system will have to start a new operation, starting with a new lower cost class. Thus, a token is placed in \( P_1 \). Then, a sequence of firings \( t_3, t_5 \) will result in having two tokens stored in \( P_1 \). The system is not safe and it can start a pipeline generation with two different distribution parameters (lowest cost class). This is a conflict and the problem can be resolved only by restarting the system. To avoid this, \( t_6 \) is enabled only if there is a request (token in \( P_6 \)) and the system is available (token in \( P_6 \)).

### 4.3. Execution of the pipelines handling the segments

Once the reading stage is completed, the actual communication can start. To execute the pipelined communication, the system control requires the following sequence of events:

- Output of segment 1 (lowest segment that handles the lowest cost task) is ready. Thus, the lowest cost messages are to arrive at the target nodes.
- Messages from segment 1 are written to the memories of the target nodes.
- Output of segment 2 is ready. The task handled by segment 2 is now handled (“moves to”) by the lowest segment 1.
- Segment 2 is free (ready to accept a task from the upper segment 3) and segment 1 now handles communication previously handled by segment 2 (these are the next messages to arrive to the target nodes).
- Output of segment 3 is ready. The task handled by segment 3 moves to segment 2.
- Segment 3 is free.
- Output of segment 4 is ready. The task handled by segment 4 moves to segment 3.
- The same pattern repeats until all \( d \) segments “move down” to segment 1 and complete communication.

As described in the previous subsection, once node \( P_6 \) receives a token, the execution stage can start. This is done by firing \( t_6 \). When \( t_6 \) fires, one token is removed from \( P_6 \) and put in all places that indicate the output of one pipeline segment is ready. Since a distribution problem has \( d \) different costs (thus, \( d \) pipeline tasks), the model requires a maximum of \( d \) segments; thus, \( d \) places receive a token by firing \( t_6 \). Figure 6(a) shows the model of the pipeline execution subnetwork (recall that place and transition numbering continue from the reading subnetwork) and Figure 6(b) shows the marking that arises by firing \( t_6 \). This marking is \( \mu_1 = (P_7, P_{13}, P_{14}, P_{18}, \ldots, P_{m+1}) \), where \( P_{m+1} \) is the place showing that the output of segment...
\( d \) is ready. Clearly, the model suggests that segments correspond with “circles” of places and transitions, which are clearly formed in Figure 6(a). Transfer between segments is implemented via firing the dual input transitions. Also, note that the circles are symmetric, with the only exception being the last segment where there is one node missing for the simple reason that there is no other segment to move down to \( d \). The dashed arrows indicate that the pattern repeats until the last segment \( d \). The places and transitions of the model are described as follows:

**Places**
- \( P_7 \): Output of segment 1 is ready
- \( P_8 \): Segment 1 busy
- \( P_9 \): Segment 2 in segment 1
- \( P_{10} \): Lowest cost communication done
- \( P_{11} \): Segment 2 empty
- \( P_{12} \): Segment 3 in segment 2
- \( P_{13} \): Segment 2 output ready
- \( P_{14} \): Segment 3 output ready
- \( P_{15} \): Segment 4 in segment 3
- \( P_{16} \): Segment 3 empty
- \( P_{17} \): Segment 4 empty
- \( P_{18} \): Segment 4 output ready
\( P_m \): Segment \( d \) empty

\( P_{m+1} \): Segment \( d \) output ready

\( P_{m+2} \): Lower segment communication continues

Transitions

\( t_8 \): Unpack messages

\( t_9 \): Write to receiving processors’ memory

\( t_{10} \): Reset segment 2

\( t_{11} \): “Move” segment 2 to segment 1

\( t_{12} \): “Move” segment 3 to segment 2

\( t_{13} \): Reset segment 3

\( t_{14} \): Reset segment 4

\( t_{15} \): “Move” segment 4 to segment 3

\( t_{16} \): “Move” segment 5 to segment 4

\( t_{17} \): Reset segment 5

\( t_n \): Reset segment \( d \)

\( t_{n+1} \): “Move” segment \( d \) to segment \( d - 1 \)

\( t_{n+2} \): Proceed communication in lower segment

The proposed model is fully symmetric and it can be used easily for any number of states. Now, we are going to perform a test for the small distribution problem with parameters \( P = Q = 9, r = 4, \) and \( s = 5 \) presented in Section 3. This test will help us understand fully the execution of the model. Based on the CPT (Table 3), there are two pipeline operations, each including four tasks \( (T_0) - T_3 \) (see Table 4). The “movement” of these tasks across the pipeline segments is shown in Figure 2. The model for this distribution would include four circles formed by places and transitions and it is shown in Figure 7(a).
Initially, there are four places having a token, so $\mu_1 = (P_7, P_{13}, P_{14}, \text{ and } P_{18})$. Thus, the only enabled transition is $t_8$. Once $t_8$ fires, a token is removed from $P_7$ and placed in $P_{10}$, resulting in $(P_{10}, P_{13}, P_{14}, \text{ and } P_{18})$. From the real system’s point of view, this describes event \textbf{completion of communications performed by task $T_0$} (11) ($T_0$ is handled by segment 1, see Figure 2). Next, only transition $t_{11}$ is the only transition enabled. When it fires, it produces $\mu_2 = (P_9, P_{14}, \text{ and } P_{18})$ (tokens are removed from $P_{10}, P_{13}$, and one token is placed in $P_9$). In the real system event, \textbf{task $T_1$ is assigned to segment 1} (2) (because segment 2 moves to segment 1). Next, only transition $t_{10}$ can fire, resulting in $\mu_3 = (P_8, P_{11}, P_{14}, \text{ and } P_{18})$. We have two events now: \textbf{segment 2 is empty} (3) and \textbf{segment 1 is now busy with $T_1$} (4). At this point, there is a synchronization issue: if $t_9$ is enabled, the same pattern will be executed continuously. In the first of this series of executions, segment 1 will deliver some messages to the source processors, but afterwards there will be no messages to deliver because it will have to wait for tasks from segment 2 that do not come since segment 2 is empty waiting for upper segments. Practically, this means that the system stays idle for as long as this repeating execution continues and if this sequence does not change, the system \textbf{reaches a deadlock} and transmission has to start from scratch. Also, if $t_{12}$ does not fire in the meantime, more than one token will be placed on $P_{11}$ rendering the system unsafe. The solution we give with this model is to enable $t_9$ only after $t_8$ fires, which, for the real system, means that segment 1 can finish another distribution task only when all the remaining tasks are assigned to segments (“move downwards” as can be seen in Figure 2). When $t_{12}$ fires, a token is put in place $P_{19}$ and this enables $t_8$. However, $t_{16}$ can fire only after $t_{10}, t_{12}, t_{13}, t_{15}$, and $t_{15}$ have fired, so all the tasks have been properly assigned to segments. That is, the model indicates that there must be a synchronization between segments (and consequently between the tasks they include) to avoid deadlocks.

Continuing the description, $t_{12}$ is the only enabled transition. Once it fires, it produces $\mu_4 = (P_8, P_{12}, \text{ and } P_{18})$. This indicates that event \textbf{task $T_2$ is assigned to segment 2} (5) (segment 3 moves to segment 2). Now, $t_{13}$ is enabled. Once it fires, the new marking is $\mu_5 = (P_8, P_{13}, P_{16}, \text{ and } P_{18})$. Now, there are two events: \textbf{output of segment 2 is ready} (6) and \textbf{segment 3 is empty} (7). Event 6 means that $T_2$ is ready to move to segment 1 after the completion of $T_2$ executed there. From marking $\mu_5$, it is obvious that only $t_{15}$ can fire, resulting in $\mu_6 = (P_8, P_{13}, \text{ and } P_{15})$. For the real system, event \textbf{task $T_3$ is assigned to segment 3} (8) (because segment 4 moves to segment 3). Now, $t_{14}$ is enabled. When it fires, the new marking is $\mu_7 = (P_8, P_{13}, P_{14}, \text{ and } P_{17})$. The two real system events are: \textbf{output of segment 3 is ready} (9) (meaning that when $T_1$ finishes from segment 1 and $T_2$ is pushed from segment 2 to segment 1, $T_3$ will move to segment 2) and \textbf{segment 4 is empty} (10). Now $t_{16}$ is enabled. When it fires, the new marking is $\mu_8 = (P_8, P_{13}, P_{14}, P_{18}, \text{ and } P_{19})$. With a token in $P_{19}$, transition $t_9$ is enabled again because all segments have moved as explained previously. When this firing occurs, there is one event: \textbf{completion of communications performed by task $T_4$} (11). Also, the new marking produced is $\mu_9$, meaning that the same sequence of firings repeats. Therefore, another transfer (task $T_3$) can be completed in the lowest segment 1 and the segments will move downwards again, repeating the same execution cycle. Next, the sequence of events produced by the model’s execution is listed (the events written boldfaced in the analysis above).

(1) Completion of communications performed by task $T_0$
(2) Task $T_1$ is assigned to segment 1
(3) Segment 2 is empty
(4) Segment 1 is now busy with $T_1$
(5) Task $T_2$ is assigned to segment 2
(6) Output of segment 2 is ready
(7) Segment 3 is empty
(8) Task $T_3$ is assigned to segment 3
(9) Output of segment 3 is ready
(10) Segment 4 is empty
(11) Completion of communications performed by task $T_0$
(12) Same pattern repeats

Based on the above analysis, it is easy to get the reachability tree of Figure 7(b). Obviously, the PN model has no deadlocks since there is no sequence of firings that can disable a transition and the pattern is proven to repeat itself. As an anti-paradigm, suppose that the at least one of the places $P_7, P_{13}, P_{14},$ and $P_{18}$, say $P_{13}$ is not initially marked. When $t_9$ fires, a token will move to $P_{10}$, but since $P_{13}$ has no token, $t_{11}$ is disabled causing a deadlock. For the real system, this means that segment 2 cannot move down to segment 1, simply because its output is not, and will never be, ready. This will halt the system causing unwanted effects (e.g. communication restart).

4.4. Advantages of the proposed model
At this point, it is important to summarize some of the advantages of the proposed model.

(a) Optimal in terms of complexity: The complexity of the model is $O(n)$, where $n$ is the number of states examined. Clearly, this is the best that can be achieved as any other method would have to examine every state, so the complexity would at least be $O(n)$.
(b) Symmetry: As stated before, the model can be used for any redistribution problem (that is, any number of states) due to its symmetry. Thus, generally, there are no applicability limitations.
(c) Precision: The model includes all the main processes involved in a redistribution problem (as described in Section 3): generation of the pipeline tasks, initialization and reading messages from memory, transferring, and writing back to memory. Thus, it is precise and models the real problem with high accuracy.

Remark When the second cycle is executed, segment 4 will be found empty (from the execution of $t_{14}$ in the first cycle). This means that segment 4 can now be used to put the lowest cost task of the next pipeline communication. In this case, when $t_{15}$ fires, the lowest task of the next pipeline can move to segment 3. Similarly, when the third cycle executes, segment 3 will be found empty (from the execution of $t_{13}$ in the second cycle). So, the lowest cost task of the second pipeline can move from segment 4 to segment 3, while the next lowest cost task can enter segment 4. However, it must be pointed out that the communication model is not designed to take advantage of the empty segments in an organized manner. Maybe, an idea would be to create and pipeline groups of classes to assure some kind of transfer homogeneity. This is a subject of future research.

5. Experiments
In this section, the accuracy of the model is verified via simulations. Based on the PN model, a small pipeline simulator (PPN simulator) was implemented to serve the purpose of this work. The simulator simply executes the well-defined sequence of processors described in Section 3.3. The simulations performed are not restricted by the assumption that there is adequate bandwidth and enough buffer space imposed for the sake of the model. Instead, different bandwidth sizes are used and simulations were performed considering the fact that buffers may or may not have enough size for the data volume carried over the network. Different scenarios are studied in terms of buffer space, bandwidth, and data volumes. The interprocessor communication is performed via all to all links. Each link has a bandwidth of $B$ MB/sec bandwidth. Considering that each node sends a number of messages at each communication step, the bandwidth available to a node would be:

$$\text{bandwidth}_p = B \times \frac{\ell}{h}$$

where $B$ MB is the link bandwidth, $\ell$ is the number of links to a node $p$, and $h$ is the average number of intermediate hops required for the communication between two processors. Assuming that there is only one link for each node and since $h$ depends on the logarithmic value of $P$, Equation 4 becomes:
To run the simulations, the theoretical block size $s$ of the target distribution must be converted to real time. By multiplying $s$ to an assumed vector size, the message size transferred between two processors at a step is produced. For example, if the target distribution block size $s = 5$ and vector size is 1 MB, then processor $p$ will send a message of 5 MB to processor $q$. Since the bandwidth is $B$ Mb/sec, it is easy to estimate the time required for the transmission. In the following, the vector sizes will also be variable. In half of the simulations ran, the buffer spaces available were considered to be, on the average, $u$% less than the maximum bandwidth. For the remaining half simulations, the buffer sizes were considered to have enough space to accommodate the data.

5.1. Scenario 1
The parameters and their values for the first scenario are as follows:

1. $r = 7$ and $s = 11$
2. Number of processors $P = 8$.
3. Vector size: ranging from 128 Kb to 2 Mb
4. Real data volumes distributed: 1408 Kb–22 MB
5. $B = 50$, so the bandwidth available is $\frac{50}{\log_2(P)}$  
6. $u$ is 0 or 40 (when it is 0, the buffer space is enough)

Figure 8(a) shows the simulation results. The bandwidth available for a processor $p$ is computed as

$$\text{bandwidth}_p = B \times \frac{1}{\log_2(P)}$$  \hspace{1cm} (5)

Figure 8. Experiments for various parameters.
128 × 11 = 1408 KB to a receiver q. Since the maximum number of sending processors towards q is 8, q can receive a maximum of approximately 8 × 1408 Kb = 11 Mb. As the vector size increases, the gap between the two lines increases.

5.2. Scenario 2
In the second scenario, the vector size ranges from 32 to 512 Kb and the number of processors is $P = Q = 16$. The parameters and their values for the first scenario are as follows:

1. $r = 7$ and $s = 11$
2. Number if processors $P = Q = 16$
3. Vector size: ranging from 32 to 512 Kb
4. Real data volumes distributed: 352 Kb–5.5 MB
5. $B = 50$, so the bandwidth available is $\frac{50}{\log_{16}}$
6. $u$ is 0 or 30

Figure 8(b) shows the simulation results. The bandwidth available for a processor $p$ is computed as $\text{bandwidth}_p = \frac{50}{\log_{16}} \approx 8.33$ MB. The upper line gives the results on the basis that the buffers have, on the average, 30% less capacity than the maximum bandwidth ($u = 30$), in this case, $\approx 5.8$ MB. The lower line assumes that the buffers have enough space to accommodate the messages ($u = 0$). For vector sizes less than 32 Kb, the buffers can accommodate the messages arrived because each sending processor $p$ sends a maximum of $32 \times 11 = 352$ KB to a receiver $q$. Since the maximum number of sending processors towards $q$ is 16, $q$ can receive a maximum of approximately $16 \times 352$ Kb = 5.5 Mb. As the vector size increases, the gap between the two lines increases. In this scenario, $m$ decreases by 10% compared to the first scenario. This means that the memory can accommodate higher percentages of the total data volumes (recall that as the number of processors increases, the available bandwidth drops off). Thus, the two lines converge more compared to the first scenario.

5.3. Scenario 3
In the second scenario, the vector size ranges from 8 to 512 Kb and the number of processors is $P = Q = 64$. The parameters and their values for the first scenario are as follows:

1. $r = 7$ and $s = 11$
2. Number if processors $P = Q = 64$
3. Vector size: ranging from 8 to 512 Kb
4. Real data volumes distributed: 88 Kb–5.5 MB
5. $B = 50$, so the bandwidth available is $\frac{50}{\log_{64}}$
6. $u$ is 0 or 10

Figure 8(b) shows the simulation results. The bandwidth available for a processor $p$ is computed as $\text{bandwidth}_p = \frac{50}{\log_{64}} \approx 6.25$ MB. The upper line gives the results on the basis that the buffers have, on the average, 10% less capacity than the maximum bandwidth ($u = 10$), in this case, $\approx 5.6$ MB. The lower line assumes that the buffers have enough space to accommodate the messages ($u = 0$). For vector sizes less than 8 Kb, the buffers can accommodate the messages arrived because each sending processor $p$ sends a maximum of $8 \times 11 = 88$ KB to a receiver $q$. Since the maximum number of sending processors towards $q$ is 64, $q$ can receive a maximum of approximately $64 \times 88$ Kb = 5.5 Mb. As the vector size increases, the gap between the two lines increases. Again, notice that $m$ decreases by another 20% compared to the second scenario, meaning that the memory can accommodate higher percentages of the total data volumes. Thus, the two lines are closer compared to the second scenario.

As an observation, one can state that the results the model produces for the three scenarios corroborate each other. In the first scenario, the local processors’ memories were not capable of storing
a good percentage of the data carried over the network. While $U$ keeps on reducing and the number of processors increases (thus, reducing the bandwidth and the data volumes distributed), the two lines shown in each of the three graphs are converging. The correctness of the results verifies the validity of the model.

6. Conclusions—future research

This paper presents a PN-based model used to verify and evaluate the performance of pipelined parallel distributions. It precisely captures the behavior of a pipeline-based parallel communication system. The model considers message scheduling and message classification, while it is deadlock and contention free. Because it is symmetric, it can easily be used for larger systems only with minor changes. This is one of its biggest strengths.

Future work can include the study of pipelined systems on certain topologies. It is interesting to check if the proposed model (perhaps with minor changes) can be used to study the performance of a data distribution over a torus or mesh network. Also, as already mentioned in the remark of Section IV, an implementation that can take advantage of any empty segments that incur during the distribution is of particular interest. An idea would be the introduction of superclasses (groups of classes).

Funding

The authors received no direct funding for this research.

Author details

Stavros I. Souravlas1
E-mail: soustav@uom.gr
Manos Roumeliotis1
E-mail: manos@uom.gr
1 Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece.

Citation information

Cite this article as: Petri net modeling and simulation of pipelined redistributions for a deadlock-free system, Stavros I. Souravlas & Manos Roumeliotis, Cogent Engineering (2015), 2: 1057427.

References


