The PN-PEM framework: a Petri Net Based Parallel Execution Model

Gustavo Wolfmann
Laboratorio de Computación
Fac. Cs. Exactas Físicas y Naturales - Universidad Nacional de Córdoba
Av. Vélez Sársfield 1611 - Córdoba - Argentina
agustavo.wolfmann@unc.edu.ar
and
Armando De Giusti
Instituto de Investigación en Informática LIDI (III-LIDI)
Fac. de Informática - Universidad Nacional de La Plata
CONICET
50 y 120 - La Plata - Argentina
degiusti@lidi.info.unlp.edu.ar

***Abstract*—This paper introduces the PN-PEM framework. It is based on the representation of an algorithm with Petri Nets. Frequently, a real algorithm needs a large Petri Net to be represented. We present a way to model an algorithm with Colored Petri Nets that simplify the model. After that, this high level model is transformed into a low level but executable model, preserving its semantics. The execution also needs other components of the framework, as the involved processors, data used and executable kernels. The combination of these elements is described in order to obtain a parallel execution. Some tests are also presented as a testbed of the framework in symmetric multiprocessors. Usability, as well as good performance, confirm the quality of the framework.**

Keywords*—Parallel Programming - Asynchronous Parallel Execution - Petri Nets - Framework.

I. Introduction

Synchronicity is a strategy to facilitate parallel programming. It is simple for programmers that all processors in a parallel program reach a point after which they complete a certain number of tasks. Nevertheless, synchronicity is a factor that is unfavorable for the overall performance of the algorithm: in a barrier, faster processors are forced to remain idle while waiting for slower ones.

The tiled algorithms emerge as a solution to the problem of load balance for dense linear algebra algorithms on multicore processors [4]. This type of algorithms is an evolution from rectangular block-based algorithms, in which data reusability was the concept to optimize. However, now the key concepts are fine granularity and asynchronicity to achieve better thread level parallelism.

Tiled algorithms divide data in square blocks that allow "out of order" computing, thus increasing the number of tasks that can be run in parallel. As with block-based algorithms, classical factorizations and updates consist in applying the proper routines ("kernels") from the BLAS [1] and LAPACK libraries [2]. Block sizes are tweaked to achieve good execution performance for the kernels involved in the algorithm.

Since the number of tasks available to run in parallel exceeds the number of processors, different tasks can be selected to define the scheduling of the parallel algorithm. Static scheduling is defined prior the algorithm execution. Common examples are left looking (LL) and right looking (RL) scheduling, which differ based on whether priority updates are on the left or on the right of the current factorization panel [11], [14]. Task scheduling plays a significant role in the performance achieved.

An asynchronous execution is complex to model and to execute. The main advantage of this model is that there are no idle processors if there are tasks that can be executed and that are waiting for a free processor. When the program has different tasks, or task with divergences, load unbalancing occurs, causing idleness in processors when execution is synchronous.

On the other hand, a coordination element must be present in the asynchronous model, to manage tasks execution. In this case, the programmer does not have absolute control over the execution because each processor behave independently. Therefore, guidelines should be provided to select tasks to execute, in a way that is independent of the progress of processing.

The "Master / Slave" model is an example of this kind of parallel programming [13]. In the same line is the "Peer to Peer" model, which has a non-centralized administration of tasks, distributed between the peers involved in the execution.

Both models, "Master / Slave" and "Peer to Peer", were designed to be run on a distributed system using message passing communication. An implementation on a multiprocessor with shared memory, should simulate that message passing model using shared variables.

By other side, Petri Nets is a mathematical and graphical model that allows to represent a concurrent process. It is a bipartite graph formed by a set of Places, a set of Transitions, and Tokens, that are found only in Places. It is bipartite because a Place can be only linked to Transitions and Transitions can be only linked to Places. Transition "firing" is the concept that allows representing the execution of the model. As many Transitions can be fired simultaneously, the concurrency of the model is represented naturally.

Since the execution of a program is a process, the representation of a parallel execution of a program by a Petri Net is easily modeled. There are many examples of representation of a program with Petri Nets [6], [10], [12], mainly oriented to the study of properties of the algorithm or to simulate its execution. But when it comes to executing the PN model on a real computer, a gap between both stages emerges. To the best

of our knowledge, nobody uses PNs model to run a parallel program.

The objective of this paper is to introduce the PN-PEM framework that allows to model a parallel program using Petri Nets and to run it without using any other tools, addressing the gap mentioned above. It is based on the research that lead to the PhD dissertation of one of the authors [20].

II. The PN-PEM Model

A. Petri Nets Context

A Petri Net (PN) is a bipartite directed graph that consists of nodes called Places and Transitions. Usually, Places represent "states" and Transitions represent "actions". The set of Places is denoted by P and the set of Transitions is denoted by T [5]. The PN has tokens, which only exist in Places, and usually represents "facts". The marking function is, $\mu : P \rightarrow \mathbb{N}$ and defines the number of tokens belonging to each Place. The set of Places whose arcs are directed to a Transition t is called Input Places of t. In the same sense, the set of Places whose arcs come from a Transition t is called Output Places of t. A Marking Vector is denoted by $M \in \mathbb{N}^{1\times|P|}$ where $M[i]$ denotes the number of tokens in the place i. Tokens are indistinguishables from each other. The initial state of the marking vector is denoted by M_0.

A transition t is said to be enabled when all its input Places have tokens. The weight of an arc $w(a)$ can be greater or equal to one. The execution of the net occurs when a transition t is "fired", moving tokens from its input Places to its output Places. When firing a transition, the Marking Vector is updated, subtracting values in the positions that represent the Input Places of t and adding values to the positions of its output Places. This net is also known as Token Petri Net (TPN).

The set of arcs A can represented by a matrix, called Incidence Matrix $D \in \mathbb{N}^{|P|\times|T|}$, where rows represent places and columns represent transitions. The values in the matrix are $w(a)$ or 0 depending on whether arc $a = (p, t)$ or $a = (t, p)$ belongs to A or not. Two Incidence Matrices are defined: D^- and D^+, the Negative and Positive Incidence Matrices respectively, representing weight valuesfor the Input and Output Places.

Colored Petri Nets (CPN) are a class of High Level Petri Nets [5] that allow modeling complex problems in a compact form. Their most notorious property is that tokens may have differences between them. The mathematical concept of domain is used to define the "color" in the CPN. In this model, each Place is related to one domain; thus, colors represent the domain of a place and tokens must be linked to a color, so as not to be fungible as in TPNs.

Two additional concepts are used in CPN. First, multisets are required to represent the stock of tokens in each Place; this is done using a $< color, quantity >$ pair. Second, the guard is a boolean expression that determines when a transition is enabled and which are the tokens absorbed from each input place. CPNs are not usually represented using matrices due to their complexity.

B. The PN-PEM process to Model the algorithm

Once the programmer has analyzed the algorithm and has determined the tasks and the number of data divisions, it must represent the algorithm in a CPN with the following conditions:

- Each task of the algorithm is represented by one and only one Transition.
- Each data argument of each task is represented by an input Place of the corresponding Transition. In other words, a Transition has as many input places as data parameters the represented routine has.
- No other Places and Transitions are defined.
- Each Transition is linked to its output Places, which are some of the input Places defined before. This represents data dependency, because the output of a Transition is used as input for another.
- The domains of each place are determined in order to represent the algorithm conditions in the net.
- The initial mark, $_0$, is defined. It has positions with zero, the ones that represent intermediate states of the algorithm, and other positions with the label of the data blocks that represent the initial values of the data used by the algorithm.

The key point in the process is to correctly define the domain of each Place and the guard functions. Since each Place represents an input parameter for a task, the domain of each Place is the naming assigned to the data division where data are partitioned for parallel execution. The domain may be restricted by the same conditions that the task imposes on each parameter. For example, in the linear algebra tiled algorithms [4], the domain of each Place should be a pair $< row, col >$, denoting the position of the block inside the matrix. The next subsections show examples of the models of two algorithms built in this way.

This high level model is usually used to represent algorithms and simulate their execution. It has the major drawback that it cannot be executed directly on a parallel computer. The overhead necessary to abstractly represent domains and function arcs is expensive in terms of high performance computing.

On the other hand, a CPN developed in this way fulfill the definition of well-formed CPNs [5]. These nets are easily transformed into TPNs, by unfolding thr CPN into a TPN. The latter has a computational implementation that is simple and light to execute.

The TPN obtained by unfolding a CPN is produced as follows:

- Each place in the CPN is replaced by a set of Places in the TPN, one for each value in the domain of the original Place, preserving the number of repetitions of each color, due to its multiset model representation, the same as each Place represents a value of a token in the CPN.
- Each Transition in the CPN is replaced by a set of Transitions in the TPN, one for each combination of

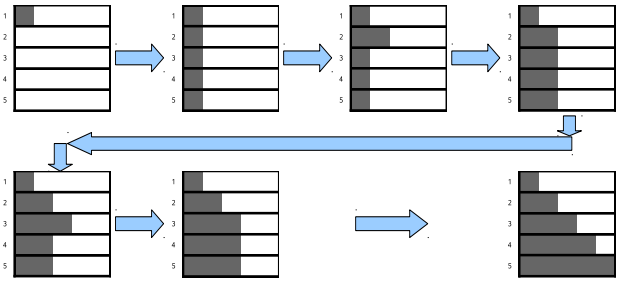

Fig. 1. Execution steps for the Cholesky algorithm, with $n = 5$

colors in the Cartesian product of the input Places of the CPN, restricted to the cases that its guard function evaluates as true. The input Places of the unfolded Transition are the corresponding Places generated above.

It should be noted that the unfolded TPN represents the algorithm because it preserves the semantics of the CPN through construction. Thus, each Transition in the unfolded represents each task, and each Place, to each data parameter.

Before continuing describing the rest of the PN-PEM framework, two sample algorithms generated like this are presented.

C. The Cholesky Factorization Algorithm

The Cholesky factorization algorithm is a classical problem of linear algebra, in which a square, symmetric and positive defined matrix A, with range r, is factorized as $A = L * L^T$, where L is a triangular matrix. The values of L are calculated with the following formulas, using the lower portion of the triangular matrix:

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} * l_{jk} \right) / l_{jj} \quad 1 \leq j < i \leq r \quad (1)$$

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2} \quad 1 \leq i \leq r \quad (2)$$

where a hard data dependency can be observed, since, to calculate the values on the main diagonal, the values on the same row are required, which, in turn, needs the values from the previous rows.

The parallel implementation used in this work is based on the tiled version of the algorithm as defined in [15], using a data division of square blocks and routines from BLAS [1] and LAPACK libraries [2] for computation. These routines are xPOTRF, xGEMM, xTRSM and xSYRK, where x can be 's' or 'd' depending on whether single or double precision data are used. Execution steps are depicted in Fig.(1), supposing a tile division of $n = 5$.

Fig. 2 shows the CPN that represents Cholesky's algorithm. It has only four Transitions and eight Places; each Transition represents one routine and each Place represents one of its parameters. The name of the Places follows the number of the block used in each operation. Color tokens are represented by

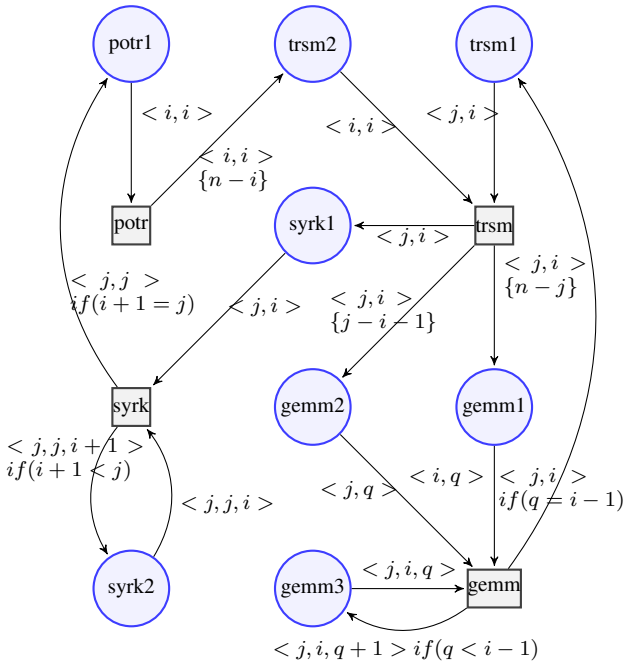

Fig. 2. Colored Petri Net that represents Cholesky's factorization algorithm.

$< x,y >$, multiset repetitions by braces $\{x\}$, and functions arcs are only Booleans of the form $if(cond)$.

The domains used in Places have the format $< x, y >$, representing the row and column of the block in the matrix, with some restrictions imposed by the algorithm. The number of tiles in which the matrix is divided is n and it is provided as a parameter. The first two columns in the Table I show the domains used for each Place. Restrictions in the domains are basically caused by the triangular shape of the matrix and the placement of the parameters used in each routine. The third parameter used in some Places responds to the necessity to preserve an order in the computations. In these cases, the operation is originally based on rectangular data blocks, which, due the tiled divisions, are square in our case.

The last column in the Table I shows the Places of an unfolding example, with the $n = 3$ for simplicity. The names of Places in the TPN follow the corresponding names in the CPN, concatenated with the color of the token that is represented. For example, syrk132, is the Place in TPN that came from Place syrk1 with color $< 3, 2 >$ in CPN, and represents the first argument in the syrk operation of the tile in the third row, second column. An example with $n = 4$ is depicted in Fig.3

The table in Fig. II shows the number of tasks in the algorithm based on the number of tile divisions. The *potr* task has a linear growth, *syrk* and *trsm* have squared growth, and *gemm*, cubic growth. The number of tiles n defines a series of stages in the processing, where each incremental value of n introduces a group of new three serial tasks. This implies that, for example, a tile division with $n = 6$ has 16 sequential tasks over the total of 56.

It is not difficult to see how fast the number of Places and Transitions in TPN grows with an increasing number of tile

TABLE I. DOMAINS FOR THE PLACES IN THE CPN IN FIG.2. THE LAST COLUMN SHOWS THE PLACES OF THE TPN UNFOLDED WITH A TILE DIVISION OF $n = 3$.

CPN Place	Domain in CPN	TPN Places
potr1	$< i, i >$ $i = 1 \dots n$	potr111 potr122 potr133
trsm1	$< j, i >$ $j = 2 \dots n$ $i = 1 \dots j - 1, j > i$	trsm121 trsm131 trsm132
trsm2	$< i, i >$ $\{n - 1\}$ repts	trsm211 $\{2\}$ trsm222 $\{1\}$
syrk1	$< j, i >$ $j = 2 \dots n$ $i = 1 \dots j - 1, j > i$	syrk121 syrk131 syrk132
syrk2	$< j, j, i >$ $j = 2 \dots n$ $i = 1 \dots j - 1, j > i$	syrk2221 syrk2331 syrk2332
gemm1	$< j, i >, j > i$ $j = 2 \dots n$ $i = 1 \dots j - 1$,$\{n - j\}$ repts	gemm121 $\{1\}$
gemm2	$< j, i >, j > i$ $j = 3 \dots n$ $i = 1 \dots j - 2$,$\{j - i - 1\}$ repts	gemm231 $\{1\}$
gemm3	$< j, i, q >, j > i > q$ $j = 3 \dots n, i = 2 \dots n - 1$ $q = 1 \dots i - 1$	gemm3321

TABLE II. REPETITION NUMBER OF EACH TASK BASED ON TILE DIVISION.

op $\backslash n$	1	2	3	4	5	6	8	10	12	15	20
potr	1	2	3	4	5	6	8	10	12	15	20
syrk	0	1	3	6	10	15	28	45	66	105	190
trsm	0	1	3	6	10	15	28	45	66	105	190
gemm	0	0	1	4	10	20	56	120	220	455	1140
total	1	4	10	20	35	56	120	220	364	680	1540
seq. tasks	1	4	7	10	13	16	19	22	25	28	31

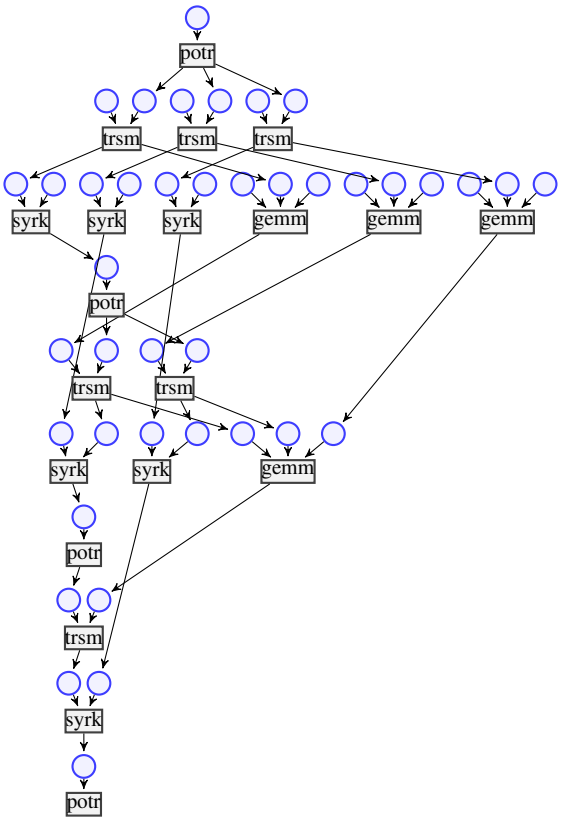

Fig. 3. Token Petri Net unfolded from the Colored Petri Net shown in Fig.2 using a tile division of $n = 4$.

divisions. It is practically impossible to show and understand its graphical representation. An example of an unfolded net with $n = 4$ is shown in Fig.3. This figure shows the existence of tasks whose dependencies are multiple and whose execution must be initiated as fast as possible, and other tasks whose execution can be delayed without affecting the ending.

In addition to facilitating the analysis of the algorithm, the key feature offered by the unfolded net is a matrix-like representation of the net. Thus, a simple pair of matrices represent the tasks, data and dependencies, are the model of the algorithm. Its execution is guided by matrix - vector operations. A framework that interprets this information and link it with real routines, is introduced in the next section. Before that, another example is presented.

D. The Merge-Sort Algorithm

To present another algorithm modeling example using PN-PEM, the merge-sort algorithm should be described. Merge-sort is a sorting algorithm that has two stages: data division stage and a stage for merging the sorted blocks. Data division is a recursive process that divides data in two halves until an individual data block is obtained. The merging stage joins together the previously divided blocks taking one and one with a lower value from the two blocks until all values are sorted [16].

A parallel implementation merges the divided blocks until no more tasks can be done in parallel, which occurs at the last step of merging. However, to improve efficiency, the data division stage continues not until a unitary value block is obtained, but rather a block reaches a minimum size. This minimum size block is sorted by some algorithm, after which the merging stage begins [9].

Three tasks are required to build the CPN model for this algorithm; they are dividing, sorting and merging. Each of these is represented by a Transition in the CPN. Considering the data disposed in an indexable support, data division is defined by $< x, y >$ segments, indicating the beginning and the end of the segment.

The division task takes one segment and produces two. The sorting task takes one segment and produces another where the data have been sorted, and the merging task takes two arguments and produces one, which is the result of joining of both initial arguments. Fig. 4 shows a graphical representation of the model. Places and Transitions names follow the criteria described in Cholesky's example.

It should be noted that the domains of the Places in the Figure takes the form of an ordered trio. The reason to add a dimension in the colors of the net is to preserve the order of the recursive data division needed at merging time. As the recursion implicitly uses a stack, this stack is simulated with in this third dimension.

The numbers in the third dimension represent the sequence of natural numbers in a binary tree beginning with one. Each value i has two children, $i*2$ and $i*2+1$, conforming a heap. In this way the ancestor of a number can be determined using the integer division by two. The rest of the integer division

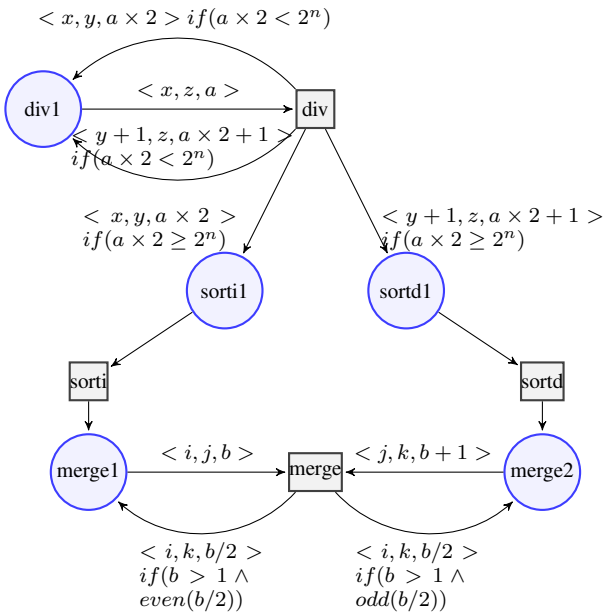

Fig. 4. Colored Petri Net that represents the Merge-Sort by blocks algorithm. The number of subdivisions is done by n. The operation of division is the usual between integers.

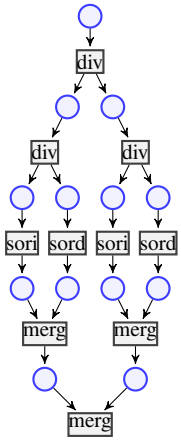

Fig. 5. Token Petri Net unfolded from the Colored Petri Net shown in Fig. 4 using a number of divisions $n = 2$.

determines the left or right position of the segment in the merging phase. The number of recursive divisions n is given as a parameter.

Fig. 5 shows the unfolded TPN for this algorithm for the elemental case of only two levels of data divisions. It can be seen that, as the number of divisions increases, the number of tasks increases to the power of two. Since the division task is only a segmentation task, it can be considered as non-computational. Thus, the total number of computational tasks is $2^{n+1} - 1$, considering only sorting and merging tasks.

The analysis of the algorithm determines that the dependency between tasks is present at the merging stages; thus, more parallel tasks will be available with a larger number of data divisions. For an efficiently parallel run, a balance between the number of processors, data locality and the number of tasks to manage must be determined.

Even though the parallel execution of this algorithm seems to be obstacle-free, a problem arises when using heterogeneous multiprocessors. In this case, the PN-PEM model will favor the execution, because, due to the existence of processors asymmetries, task assignment to processors is complex to define statically.

There are two aspects to note in relation to the modeling of an algorithm using PN-PEM. First, it allows analyzing the parallel algorithm independently of the processors that execute it, which in turn allows drawing conclusions about its structural parallelism and the factors that need to be taken into account when scheduling tasks. Secondly, from the point of view of the Petri Nets theory, the execution of the net with its initial Mark Vector should end without any tokens at any Place. If the execution is infinite or ends with tokens, it means that the model is incorrect, which results in infinite execution or incomplete data computations.

Next section introduces the execution framework, explaining how the PN-PEM model executes an algorithm represented by Incidence Matrices and a Mark Vector.

III. The PN-PEM Framework

The unfolded TPN bluit in the previous section has the advantage that can be easily represented by the Incidence Matrix. Thus, the algorithm can be represented by two matrices of integers. This section introduces the executable section of the model, the one that takes the Incidence Matrices and runs the represented algorithm.

The PN-PEM framework is a model for the parallel execution of programs, that, instead of using low level instructions or compiler directives to indicate the parallel sections of a program, uses the Incidence Matrices as parameter of the parallel program execution. In addition to the matrices, others elements are necessary for the execution.

The PN-PEM is defined as a tuple:

$$PEM = (P, T, D^-, D^+, M, M_0, S, \tau, \delta, \Pi, \Gamma, \chi) \quad (3)$$

where:

- P is a finite set of Places P_i.
- T is a finite set of Transitions T_j.
- D^- and D^+ are the negative and positive Incidence Matrixes of the TPN.
- M, is the Mark Vector for Places.
- M_0, is the initial marking for the net.
- S is the set of tasks that the algorithm must to execute. They are obtained from the previous analysis of the algorithm.
- τ is a function from the set of transitions T to the set of task S, $\tau : T \rightarrow S$. It associates each transition with a task.
- δ is a function from the set of places P to the set of data divisions, which associates each place with a data block.

- Π is a finite set of Processors Π_i. A processor Π_i is an object capable of executing the tasks associated to each transition by calling a compute kernel.
- Γ is the set of functions γ_i, one for each processor Π_i.
- γ_i is a function, that asociates a task $s \in S$ with a kernel to execute. It defines the kernel to be executed when the Transition is fired, and links the PN-PEM model with the execution of the algorithm. Each processor Π_i has its own function γ_i, allowing different processors to solve the execution of the tasks s in their own way.
- χ is a Boolean variable that represents the state of a mutual exclusion mechanism over M that allows each Π_i to update M securely.

and the initial state of the PN-PEM is defined by marking $M = M_0$, and $\chi = true$, the exclusion is free.

For the parallel execution each processor runs a task independently of the rest. Data dependency is guaranteed by the Petri Net model when an output Place acts as an input Place in a possibly different Transition. This produces a strong effect in the model that eliminates completely the need to introduce synchronizations in the parallel execution.

The number of enabled Transitions can be lower or higher than the number of idle Processors. As a result of this, there may be idle Processors with no Transitions to fire or enabled Transitions waiting for a free Processor. In the first case, execution speedup will be poor, which should be avoided. In the second case, the Processor should select the most appropriate Transition to fire. Thus, a criterion should be defined for each processor to select the Transition to fire when more than one is available.

Since many processors run in parallel, there may be two or more of them attempting to read or write simultaneously on the Mark Vector M. Concurrent writes on the vector may produce an incorrect state of the algorithm execution. To avoid this situation, a Mutual Exclusion (mutex) mechanism is used. In this sense, Processors work serially when they are selecting the next Transition to fire, waiting for the mutex enabled condition.

The Pseudo-code of the PN-PEM execution algorithm of each Processor is presented in Fig. 6. Each Π_i Processor runs in parallel with the other of Processors. In a round-robin fashion, each idle Π_i Processor tries to hold the mutual exclusion mechanism. If it succeeds, it searches for a task to execute based on the state of the TPN. To determine which Transitions are enabled, only simple linear algebra operations are needed. Thus, if we call D_j^- and D_j^+ the j-th column (transition) in D^- and D^+ respectively, the j-transition is enabled if the vectorial subtraction $M - D_j^-$ does not have a negative value, which implies that all the input places of the j transition have tokens. Function h represents this:

$$h(j, D^-, M) = \begin{cases} 1 & \forall k = 1 \dots p : (M - D_j^-)_k \geq 0 \\ 0 & \textit{else} \end{cases}$$

and when computing h for all the columns, all enabled Transitions that are ready to be fired are determined.

```
While main algorithm has not finished
  If can hold the mutual exclusion
     Compute h function
     Select one task to execute
     Update M by absorbing tokens
     Free the exclusion
     Task execution
     Inject tokens in M
  Else
     Delay
  Endif
End
```

Fig. 6. Pseudo-code of the task selection algorithm.

To select the task to be executed (fourth step), a selector is needed. In run-time, each processor uses a value function that is applied to the set of enabled transitions, selecting the transition with highest valuation, T_k. The valuation function is a key factor for the parallel processing performance, because to select to most appropriate task, it should be adequate for each type of processor and algorithm.

PN-PEM is very similar to Timed Petri Nets [19]. Both share the concept that firing a Transition is not instantaneous. There is an interval time between the start and the end of the firing. The same as in PN-PEM, the firing action represents the execution of a task, but the difference is that in PN-PEM firing is not autonomous once the transition is enabled, as it is in Timed Petri Nets. An idle Processor is responsible to fire the Transition selected among all the enabled ones.

Steps 5 and 8 of the pseudo-code algorithm represent the evolution of the execution. As in Timed Petri Nets, tokens are absorbed and injected at two times, t_0 and $t_0 + \Delta_k$. In Step 5, tokens from the input Places of T_k are absorbed when the Transition is fired, and, in Step 8, they are injected into their output Places. Both steps updete the Mark Vector M with simple linear algebra operations:

$$\begin{aligned} M' &= M - D_k^- && \text{in 5 at } t_0 \\ M''' &= M'' + D_k^+ && \text{in 8 at } t_0 + \Delta_k \end{aligned} \tag{4}$$

Once the tokens are absorbed in Step 5 and the exclusion is released, the task is executed in Step 7 using the function γ_i. Also, the data blocks needed are selected by the δ function which uses the input places of the selected transition to get the appropriate blocks.

Mapping settings between Transitions and Places with kernels and data blocks should not be unique for all processors; this is the key to adapting the framework to a heterogeneous system. For example, in a multicore - multigpu system, each type of processor has its own mapping setting, both for routines to execute as well as for data. In the first case, the processor heterogeneity is solved. In the second, a different granularity of data for each type of processor may be the solution, highlighting an absence of synchronicity in both cases.

Since a framework is an incomplete software artifact that a programmer should complete with its own pieces of code in order to obtain a specialized program [7], is important to

highlight which are the elements that the programer of PN-PEM should provide. They are the Incidence Matrices, the set of Places and its mapping with the data block related (δ function values), the set of Transitions with the related tasks (τ function values), the set of processors (the Π set), and the kernels to be run in each processor (the Γ set).

To achieve high performance using the PN-PEM framework, two more elements, not included in the formal definition of the framework, have to be provided by the programmer. These are the configuration of each processor object (each Π_i), and the valuation function used to select the task to be run when there is more than one task available.

The configuration of each processor Π_i is formed by processor's set of cores. This variable allows configure a two-level partition of the cores, with a logical high level division, and a lower level that allows using affinity when more that one core is used. Also, the configuration can specify if the processor object is dedicated to control a coprocessor, like a GPGPU or a Xeon Phi board.

The valuation function also helps achieving high performance. It is the element that allows implementing a dynamic scheduler. Since each of the Π_i objects can be different, each of them has its own valuation function. This function evaluates the set of available Transitions and defines the one with high priority to be executed. Thus, the result of the valuation can be calculated using as parameters to the state of the processing, the set of available Transitions, and the properties of the physical processor involved. The dynamics of the scheduler is a consequence of these characteristics, and because of this, the response can differ depending on the state of the factors considered at execution time.

Finally, as a summary of framework utilization, a programer should complete it with the code of the kernels that implement the actions of each Transitions, the code that manage data and its partitions, and the code of the valuation function described above for each processor.

This section has introduced the executable side of the PN-PEM framework, completing its presentation. The next section presents some tests done using the PN-PEM framework.

IV. Experiments

This section describes the test results carried out using the Cholesky algorithm running on a Symmetric Multiprocessor (SMP) machine built with Intel processors. It must should be noted that these experiments are conceived as a testbed for the PN-PEM framework.

The SMP used is configured with two Intel Xeon E5-2680 chips, each of them with eight cores. Each core has one AVX (Advanced Vector Extensions) unit that uses 256-bits registers and can perform addition and multiplication operations simultaneously over these registers with 32 or 64-bits groups. Since the selected algorithm makes an intensive use of AVX operations, sixteen processors were considered, one for each AVX unit.

The PN-PEM configuration of the parallel hardware may vary in the number and configuration of the two-levels processor division, from only one Π_i object with sixteen cores to sixteen Π_i logical processors composed each one by only one core. This adaptability is used to test which combination offers the best performance.

TABLE III. Time in seconds and flops in GFlops for tests with ranges 24000 and 48000, using 16 threads with double level division, $\Pi_i\times$ threads (1x16,2x8,4x4,8x2,16x1), tile division $n = 12$.

Π_i	1x16		2x8		4x4		8x2		16x1	
rng	secs	flps	secs	flps	secs	flps	secs	flps	secs	flps
24K	11.7	392	10.2	453	9.0	513	8.4	549	9.52	484
48K	69.7	529	62.7	588	59.9	616	60.3	611	71.0	519

Tests were run on the Intel Composer 2013 suite, which includes the MKL BLAS/LAPACK implementation. PN-PEM has two requirements in relation to the compiler / libraries to be used; nested parallelism, needed for the two-levels division, and core affinity, to configure each Π_i internally. Both requirements are fulfilled by the compiler selected. Single-precision floating point was used for these tests.

Table III shows the most representative results using the system described below. A tile division of twelve divisions by side, $n = 12$, was used. The ranges of the matrices tested were 24000 and 48000, and the grouping of the physical cores is configured by all the possible combinations of $< \Pi_i \times threads >$ with the sixteen AVX units. The valuation function used was the same for all processors, prioritizing the task with more pending stages in its path to the end, a close concept to the "critical path".

1×16 division means that all tasks are done sequentially by the framework and the parallel execution of each task is derived to the library. On the other hand, 16×1 implies that there are many tasks running in parallel, each task using a sequential implementation of the library. The best result, 616 gflops, is achieved by a 4×4 division which means a combination of parallel tasks run on four logical processors, each of these, composed by four internal threads. Taking into account that the Rpeak of the machine is 691.2 gflops, a near-optimum result was obtained, which evidences a very good management of the parallel tasks done by the framework.

Due to the editorial space limitations, in this work were included only tests for the Cholesky algorithm using a SMP. There were also tests for the same algorithm using other SMP computer or heterogeneous multiprocessors and for different algorithms, as for merging. The space required for a complete topic presentation of these test exceeds the allowed limits, and are left to next papers. Nevertheless, we can anticipate in order to remark the usability of the framework, that its ability to adapt to changes in the algorithm or in the processors, was significant to reach excellent results.

V. Related Works

Iordache et.al. [12] use Petri Nets to model algorithms from the control point of view. They use TPNs to model, with Transitions representing tasks and Places representing data or control facts. Scheduling tasks in a parallel system is barely mentioned.

The dynamic scheduler developed is related to the one described by Quark [21]. In this work, data locality is given

priority over parallel task availability. In PN-PEM, task availability is prioritary to the locality. StarPU is a runtime system developed at the INRIA institute that launches tasks in parallel over a set of processors, using a dynamic scheduler [3]. It is based on kernels provided by the user that implement the appropriate solution for each processor. The scheduler uses estimated times to select the task to execute. The definitions of tasks, dependencies and data partition are provided within the code. Any change involves recoding, which is a drawback.

XKaapi is another runtime system developed at INRIA that launches tasks in parallel [8], with a different approach: it is based on compiler directives introduced in the source code that define the tasks that are to be run in parallel. It uses a dynamic scheduler following a FIFO order, and does not consider any other optimization factor. In addition to the differences noted above, none of the previous works are designed to be used on heterogeneous systems.

Shetti et.al. implements the HEFT (Heterogeneous Earliest-Finish-Time) scheduling algorithm in a CPU-GPU environment [17], which is similar to the scheduler implemented in this framework. It has the disadvantage of assigning task priorities before running them. Tomov et.al. [18] introduce the concept of "critical path" for scheduling of tasks in hybrid systems, but they present a static division of type of tasks to be executed by each type of processor.

VI. Conclusions

There are several points to highlight in relation to the research and development of the PN-PEM framework:

- It has been shown that Petri Nets not only offer good properties to model concurrent systems, but they are also a basis for parallel execution. It is not easy to manage the parallel execution of hundreds or even thousands of tasks, but this tool proved successful in doing so.
- Using CPN to model an algorithm allows analyzing its parallel structure and brings information about its possibilities and limitations, which are used to improve parallel performance.
- The framework can be adapted to run on different SMPs or even on Heterogeneous machines. It should be noted that the model of the algorithm remains unchanged, differing only in the configuration of the processors, the kernels and the valuation function to select the most appropriate task for each type of processor.
- In addition to theoretical facts, the framework is also able to execute in parallel and achieve in a real performance very close to its theoretical limit. Asynchronicity and affinity are key for this.

Future work will focus on implementing the framework and analyzing its execution impact in a distributed memory architecture. Also, a domain specific language will be developed for facilitate the representation of algorithms in the framework, helping, by example, in the process of unrolling or in the generation of the Incidence Matrices. Framework internal elements tweaking will also be studied, the same as using sparse matrices to represent the TPN.

References

[1] Basic Linear Algebra Subprograms Technical Forum Standard. Technical report, University of Tennessee, 2001.

[2] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[3] C. Augonnet, S. Thibault, and R. Namyst. Starpu: a runtime system for scheduling tasks over accelerator-based multicore machines. Technical Report 7240, INRIA, Mar. 2010.

[4] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report 191, LAPACK Working Note, Sept. 2007.

[5] M. Diaz. *Petri Nets: Fundamental Models, Verification and Applications*. ISTE Ltd - John Wiley & Sons, Inc., London, Hoboken, 2009.

[6] Z. Ding, H. Shen, and J. Cao. Parallel computation of continuous petri nets based on hypergraph partitioning. *The Journal of Supercomputing*, 62(1):345–377, 2012.

[7] M. E. Fayad, D. C. Schmidt, and R. E. Johnson. *Building Application Frameworks: Object-oriented Foundations of Framework Design*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[8] T. Gautier, J. V. Ferreira Lima, N. Maillard, and B. Raffin. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Boston, Massachusetts, États-Unis, May 2013.

[9] A. D. Giusti, M. R. Naiouf, F. Chichizola, and L. C. D. Giusti. Dynamic load balance in parallel merge sorting over homogeneous clusters. In *19th International Conference on Advanced Information Networking and Applications (AINA 2005), 28-30 March 2005, Taipei, Taiwan*, pages 219–222, 2005.

[10] S. Haddad and J. Pradat-Peyre. New efficient petri nets reductions for parallel programs verification. *Parallel Processing Letters*, 16(1):101–116, 2006.

[11] A. Haidar, H. Ltaief, A. YarKhan, and J. Dongarra. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. Technical Report 243, LAPACK Working Note, Mar. 2011.

[12] M. Iordache and P. Antsaklis. Petri nets and programming: A survey. In *American Control Conference, 2009. ACC '09.*, June 2009.

[13] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[14] J. Kurzak and J. J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. Technical Report 178, LAPACK Working Note, Sept. 2006.

[15] H. Ltaief, S. Tomov, R. Nath, P. Du, , and J. Dongarra. A scalable high performant cholesky factorization for multicore with gpu accelerators. Technical Report 223, LAPACK Working Note, Nov. 2009.

[16] R. Sedgewick and K. Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[17] K. R. Shetti, S. A. Fahmy, and T. Bretschneider. Optimization of the heft algorithm for a cpu-gpu environment. In *Proceedings of the 2013 International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '13, pages 212–218, Washington, DC, USA, 2013. IEEE Computer Society.

[18] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[19] J. Wang. *Timed Petri Nets: Theory and Application*. The International Series on Discrete Event Dynamic Systems. Springer US, 1998.

[20] A. G. H. Wolfmann. *PEM - Modelo de Ejecución Paralela Basado en Redes de Petri*. PhD thesis, Fac. de Informática - Universidad Nacional de La Plata, Apr 2015.

[21] A. YarKhan, J. Kurzak, and J. Dongarra. Quark users' guide: Queueing and runtime for kernels. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011.