Parallel Asynchronous Modelization and Execution of Cholesky Algorithm using Petri Nets

Gustavo Wolfmann
Laboratorio de Computación
Fac. Cs. Exactas Físicas y Naturales
Universidad Nacional de Córdoba
Av. Vélez Sársfield 1611 - Córdoba - Argentina
gwolfmann@efn.uncor.edu

Armando De Giusti
III LIDI
Fac. Informática
Universidad Nacional de La Plata
50 y 120 - La Plata - Argentina
degiusti@lidi.info.unlp.edu.ar

***Abstract*—PDPTA 2013 - Parallelization of algorithms with hard data dependency has a need of task synchronization. Synchronous parallel versions are simple to model and program, but inefficient in terms of scalability and processors use rate. The same problem for Asynchronous versions with elemental static task scheduling. Efficient Asynchronous algorithms implement out-of-order execution and are complex to model and execute. In this paper we introduce Petri Nets as a tool for simplifying the modeling and execution of parallel asynchronous versions of this kind of algorithms, while using an efficient dynamic task scheduling implementation. The Cholesky factorization algorithm was used as testbed. Simulations were carried out as a proof of concept, based on real execution times on GPGPU's, and have shown excellent performances.**

Keywords*—*Petri Net Modelization - Asynchronous Parallel Execution - Dynamic Task Scheduling - Cholesky Factorization Algorithm.

I. Introduction

The fork-join parallelization model is a natural step from a sequential to a parallel version of an algorithm. An important drawback is the insertion of synchronization points in the algorithm which compels all the processors involved in the execution to wait in idle state, the slowest. This causes poor performance and scalability in those algorithms in which the task loads of each parallel thread differ, which is typical of algorithms with data dependency [1]. QR, LU and Cholesky factorizations are algorithms with this type of problem.

Tiled algorithms emerge as a solution to the problem of load balance for dense linear algebra algorithms on multicore processors [2]. This type of algorithms are an evolution from rectangular block-based algorithms, in which data reusability was the concept to optimize. Tiled algorithms presents, as many LAPACK algorithms do, two fundamentals steps of the algorithm: panel factorization and trailing submatrix update. 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 to computing "out of order", thus increasing the number of tasks that can run in parallel. As with block based algorithms, factorizations and updates consist in applying the proper routines ("kernels") among the operations defined in the BLAS [3] and LAPACK libraries [4]. Block sizes are tuned to achieve good performance in the execution of the kernels involved in the algorithm.

The major difference between block and tiled algorithms is that the former are synchronous, whereas the later are asynchronous. The difference is well shown graphically in [5]. Asynchronicity and fine granularity make it possible for many tasks to run in parallel. "Out of order" means that while one processor computes a factorization, the others can simultaneously compute updates.

Since the number of tasks available to run in parallel exceeds the number of processors, it is possible to do different selection of tasks, in order to define the scheduling of the parallel algorithm. Static scheduling are those defined prior the algorithm execution. Common examples are the left looking (LL) or right looking (RL), which differ according to whether priority updates are on the left or on the right of the current factorization panel [1], [6]. Both algorithms are shown inf Fig. 1 and 2, and have in common that they are fork-join synchronized.

Another known technique of static scheduling is *look ahead*. As LL and RL, it is based on performing panel factorization in one thread while the remaining update submatrix from previous stages is done by others threads. It has been observed that LL and RL are the extreme points of a wide spectrum of possibilities of task selection, acting in a parametrized way *look ahead* as a path for going from one point to another [1]. All alternatives generate bubbles of idleness in the algorithm due their static nature.

Directed Acyclic Graphs (DAG) have been used to model

```
do step=1:bl_nu
  do i=1:step-1
    syrk step,i
  end
  potr step
  do j=step+1:bl_nu
    do k=1:step-1
      gemm step,k,j,k
    end
    trsm j,step
  end
end
```

Fig. 1. Left looking Cholesky

```
do step=1:bl_nu
  potr step
  do i=step+1:bl_nu
    trsm i,step
    syrk i,step
  end
  do j=step+1:bl_nu-1
    do k=j+1:bl_nu
      gemm j,step,k,step
    end
  end
end
```

Fig. 2. Right looking Cholesky

the algorithms, with the vertex representing the tasks and the edges, the dependency among them. The graph is also known as Dependency Graph. Asynchronous execution is helped by the use of DAG's to control the dependency of tasks. The DAG is mainlly used by the scheduler to select the next task [2].

Dynamic scheduling is introduced to improve static scheduling, by selecting the task on run time according the availability of free processors and enabled tasks. This type of scheduling are aimed at preventing the existence of the stalled points of static schedulers. However, they are complex and cause overhead in the algorithm execution [6].

Hogg's research shows no significant advantage in using dynamic or static schedulers [7]. He also uses a DAG to model the algorithm and the scheduling control. Concurrency control and DAG implementation generate an overhead that seems to consume the improvements of the dynamic scheduler.

Also in the line of dynamic scheduling, LAWN 243 [6] introduces the use of the "locality" parameter to help the scheduler dynamically select the next task to be assigned to a processor, according the previously used data. Improvements in parallel execution depend on the type of algorithm (LL or RL), the size of the DAG's window resident in memory and the number of tiles into which the matrix is divided.

At the best of our knowledge, all dynamic attemps are based on DAG, which are good to represent the structure of the algorithm, but not for the execution and the scheduler. Both are implemented in an ad-hoc, sophisticated style, without parallel execution modelization.

A key factor to achieve a performing parallel algorithm, is to minimize processor idle time due to synchronization. Asynchronous execution is a big step in this path. Scheduling is another. Tiled algorithms improve the parallelism of an algorithm by increasing the number of tasks. The drawback lies in the complexity of managing a large number of parallel asynchronous tasks. The lack of a model for this results in complex or pre-developed libraries implementations [6], [8].

Our research has two main objectives:

- To model the structure and parallel execution of dense linear algebra algorithms with a simple tool.
- To improve performance by minimizing processor idle time through the use of dynamic scheduler

The second objective follows the first: with a simple model, dynamic overheads decrease and the scheduler can perform an adequate selection without loss of performance.

Petri Net is the formalism chosen to represent the algorithm. Its capability to represent parallel processes is known. A few additions to this well-known formalism are enough to achieve our objectives. As a "proof of concept" we develop a simulation tool to represent and execute the Petri Nets, which simulates different running parameters.

Cholesky factorization was chosen as a testbed algorithm. We follow the kernels and DAG representation used in tiled algorithm as defined in [9]. These kernels are xPOTRF, xGEMM, xTRSM and xSYRK, where x can be 's' or 'd' depending on whether single or double precision data are used.

II. Petri Net Model of Parallel Algorithms

A Petri Net (PN) is a bipartite directed graph consisting of Places and Transition nodes. Usually, Places represent "states" and Transitions "actions". Arcs always link a Place to a Transition (acting as input) or vice versa (acting as output). There are tokens, which only exist in Places, and represents "facts". The overall state evolves when a transition is "fired", moving tokens from input places to output places. A transition can be fired when all input places have enough tokens [10]. This net is also known as Token Petri Net (TPN).

Petri Nets are used to model the algorithm, with operations (kernels to execute) represented by Transitions and data represented by Places. Input parameters are represented by arcs that go from Places to Transitions, and operations results, by arcs from Transitions to Places.

Petri Nets can model the algorithm dependencies, and can also describe the execution by means of the firing of Transitions. The following subsections explain how the net is used for both purposes.

A. *Coloured Petri Net*

Coloured Petri Nets (CPN) are one type of the many defined as "High Level Petri Nets". The major difference with TPN is that tokens have different values ("colours") from a domain. This permits to model with a high level of abstraction. Here, transitions are enabled by having not only enough tokens in their input places but also from the "color" defined. CPN definition is taken from from [10], [11].

Coloured Petri Nets permit to model complex nets at high level in a simple manner. DAGs of task dependencies with many blocks divisions are difficult to understand due to their large number of nodes (see Fig. 10 of LAWN 243 [6]). To model tiled algorithms with CPN, the main domain used to define tokens is tile position, represented by the row-column pair.

The strategy to model the algorithm is:

1) Each operation is represented by one transition
2) For each transition, there are as many input Places as data blocks parameters are involved in the operation.
3) No more places or transitions are used.
4) Output arcs represent data dependency.

To specify conditions in places, we extend or restrict the tile-block domain. Also, multisets are used to represent repetitions of blocks, and function arc expressions, to limit token flowing [11].

Fig. 3 shows the CPN that represents the Cholesky algorithm. It has only four transitions and eight places, according to the strategy suggested. The name of the places follow the number of the block used in each operation. Color token is represented by $< x,y >$, multiset repetitions by braces $\{x\}$, and functions arcs are only booleans of the form $if(cond)$.

In each place, the domains used are:

For potr1 and trsm2, the domain is $< i, i >, i = 1 \dots n$.

For trsm1, syrk1, and gemm1, the domain is $< j, i > j = 2 \dots n, i = 1 \dots j-1, j > i$

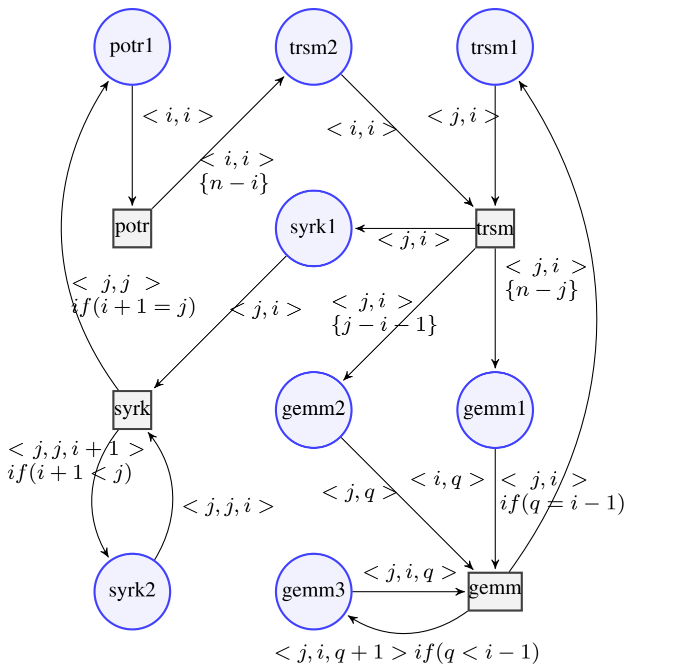

Fig. 3. Coloured Petri Net that represents Cholesky factorization algorithm.

For gemm2 the domain is $< j,i >, j = 3 \dots n, i = 1 \dots j-2, j > i$

For syrk2 the domain is $< j,j,i >, j = 2 \dots n \wedge i = 1 \dots j-1 \wedge j > i$

For gemm3 the domain is $< j,i,q >, j = 3 \dots n, i = 2 \dots n-1, q = 1 \dots i-1 \wedge j > i \wedge i > q$

The inital places mark is:

- In potr1: $< 1,1 >$
- In trsm1: $< i,1 >, i = 2..n$
- In syrk2: $< i,i,1 >, i = 2..n$
- In gemm3: $< j,i,1 >, j = 3..n, i = 2..j-1$

In this way, a tiled algorithm is generically defined by a CPN, and any consideration in the number of tiles is dispensed with, it being only a parameter for domain definition. Its simplicity and facility to analyze the parallel algorithm are highlighted.

Nevertheless, CPNs are not used to execute the algorithm. The overhead necessary to abstractly represent domains and function arcs is expensive in terms of high performance computing. On the other hand, the CPN developed in this way fulfill the definition of well-formed CPNs [10]. This type of nets are easily transformed to a TPN, which have a computational implementation that is simple and light to execute.

B. Token Petri Net

The previous section shows the facility to model a parallel algorithm with a CPN and the procedure used to define places and transitions. The resulting net is easily unfolded to a TPN.

To unfold a CPN we follow the steps defined in Diaz [10]. Each Place P_j in a CPN has a Domain $D(P_j)$ associated with it, and is unfolded to generate as many Places in TPN as is the cardinality of $D(P_j)$ in the colored Place. The repetitions from the bag that represent the Place must be respected. Thus, each Place in TPN is associated with an unique value from the pairs (color, place) in CPN and repeated according the bag pair cardinal.

The unfolding for Transitions is similar: for each Transition in CPN there will be generated as many Transitions in TPN as the cardinal of the Cartesian Product of all its input Places in the CPN, respecting the cardinal of the bag in each Place. Each Transition in TPN is associated with a unique value from tuples of the Cartesian Product, repeated as the respective cardinal of the bags of each input place.

Input Arcs in CPN is unfolded to TPN from the corresponding unfolded Place / Transition in TPN. The same occurs for output arcs, with reference to the condition of the guard function.

Table in Fig. 4 show an unfolding example for Places from CPN to TPN, for the case of 3×3 tiles divisions. The names of Places in TPN follow the respective name in CPN, concatenate 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 syrk operation of the tile in third row, second column. Graphically, the unfolded TPN of the example is shown in Fig. 5.

It is not difficult to see how fast the number of Places and Transitions in TPN grow with an incresing number of tile divisions. It is practically impossible to show and understand its graphical representation. However, the matricial representation is elementary and easy to use. The importance of unfolding is that a TPN can be represented with two matrices and one vector of natural numbers, and that elementary matrix - vector operations models the execution of the net.

The structure of TPN net can be represented by Negative and Positive Incidence Matrix (NIM / PIM). Both have dimension $p \times t$, where p is the number of Places and t is the number of Transitions. Each position in the matrix represents the relation between a pair place/transition, which is the equivalent of an arc between them in terms of graph theory. A position with zero represents absence of arc. A positive value represents the number of tokens that will be absorbed / injected by the transition depending on Negative or Positive case, if the transition is fired.

Token existence in Places is represented by a Mark Vector (MV). It has dimension $1 \times p$, and values are also naturals numbers. Values represents the number of tokens that exists in the respective Place.

The matricial representation highlights the facility to compute enabled transitions and to fire them. We call NI_j^- and PI_j^+ the j-th column (transition) in NIM and PIM respectively. By computing $MV - NI_j^-$, if the result has no negative values, MV has enough tokens in the input Places of j-transition, and thus can be fired.

Computing the difference for all the columns, determines all the transitions that are enabled to fire. By construction,

Place in CPN	Domain in CPN	Places in TPN
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	$< i, i >$ $\{n - 1\}$ repetitions	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	$< j, i >$ $j = 2 \dots n$ $i = 1 \dots j - 1, j > i$ $\{n - j\}$ repetitions	gemm121 $\{1\}$
gemm2	$< j, i >$ $j = 3 \dots n$ $i = 1 \dots j - 2, j > i$ $\{j - i - 1\}$ repetitions	gemm231 $\{1\}$
gemm3	$< j, i, q >$ $j = 3 \dots n$ $i = 2 \dots n - 1$ $q = 1 \dots i - 1$ $j > i > q$	gemm3321

Fig. 4. Unfold example for Places for the Coloured Petri Net in Fig.3 supousing only 3 × 3 tiles divisions.

Places of the unfolded TPN have only one transition to act as input. That guarantees the no competition of enabled transitions for input tokens, and that all enabled transitions can be fired simultaneously.

To modelize the net execution, an additional function is defined in order to compute the set of transitions enabled to be fired. The function is $h : \mathbb{N}^{1\times p} \times \mathbb{N}^{p\times t} \rightarrow \mathbb{N}^{1\times t}$, which has parameters M y NI^-, and its result values are:

$$h(j) = \begin{cases} 0 & \text{if } (M - NI_j^-) \text{ has negatives values} \\ 1 & \text{if } (M - NI_j^-) \text{ else} \end{cases} \quad j = 1 \dots t$$

then h positions with value 1 reference to transitions enabled to be fired.

Firing all enabled transitions defines a new mark for Mark Vector (MV'):

$$MV' = MV - h \times NIM^t + h \times PIM^t \tag{1a}$$

III. Execution Model

DAGs can model dependencies of tasks in a parallel algorithm, but they do not modelize the execution. Petri Nets have implicit modelization of execution: by representing tasks as Transitions, all enabled Transitions are those that can be executed.

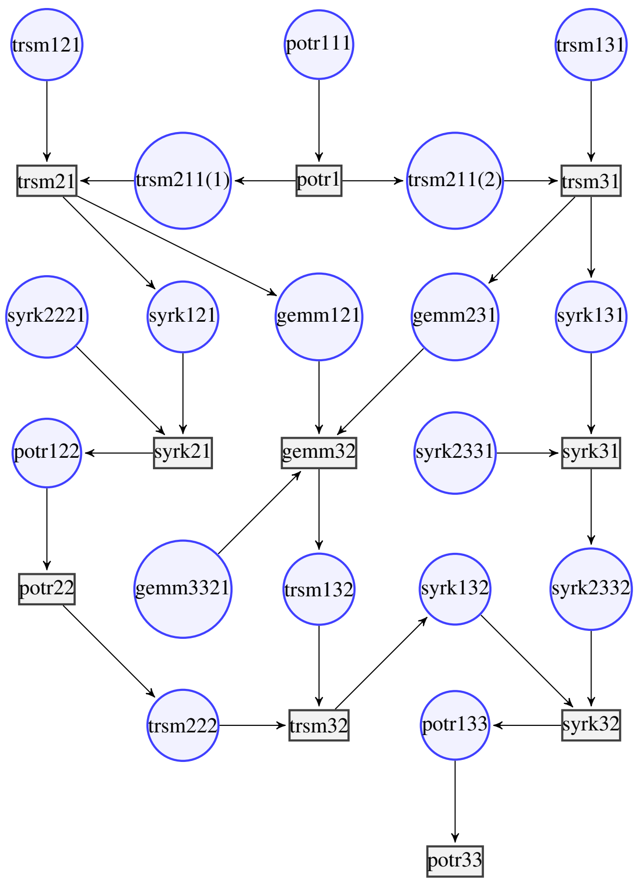

Fig. 5. Token Petri Net unfolded from the Coloured Petri Net in Fig.3 suposing only 3 x 3 tiles divisions.

Nevertheless, TPN is not sufficient to model the execution of a parallel algorithm. It has no information about the running time of a task, and has no limit about the number of processors that execute the task.

To solve the problem of execution time, we use Timed Petri Nets (TiPN) [10]. They have an important feature, the representation of the time in Transitions. By adding a delay between the time in which tokens are absorbed from input places and the time in which tokens are injected in output places, transitions can represent the notion of execution time.

Firing a transition k in TiPN implies an MV update in two times:

$$MV' = MV - NI_k^- \qquad \text{in } t_{ini} \tag{2a}$$

$$MV'' = MV' + PI_k^+ \qquad \text{in } t_{ini} + \Delta(T_k) \tag{2b}$$

where t_{ini} is the initial firing time, and $\Delta(T_k)$ is the execution time of task T_k.

To solve the problem of the availability of many processors, we define an execution model. The model consists of a set of processors and one TiPN that represent the algorithm and its dependencies as we have used along this paper. Each processor knows how to do the task that each transition represents. The TiPN is shared by all the processors. Each processor

```
While main algorithm not finished
   If can hold the mutual exclusion
      Compute function h
      Define the task to do
      Update MV by absorbing tokens
      Free the exclusion
      Task execution
      Inject tokens in MV
   Else
      Delay
   Endif
End
```

checks the TiPN state to select a task to do, from all enabled transitions. To prevent multiple selection of the same task, a mutual exclusion mechanism is added to the TiPN.

Each processor executes the following pseudo-code parallel execution algorithm:

Details of processor execution pseudo-code:

- Main algorithm is the represented by the Petri Net.
- The exclusion is hold until the tokens are absorbed from input places, before the processor begins the task execution. No colission is produced by tokens injection.
- When more than one transition is enabled, a selection policy must to placed.
- A delay is introduced if the processor can't hold the exclusion to avoid starvation.

The overhead introduced by the parallel execution is defined by three factors. First, the mutual exclusion mechanism, the execution of which uses few clocks cycles. Second, the integer matrix and vector operations, which are highly optimized to run in milliseconds in today processors. Third, the selection policy must be guided by balancing between selection algorithm and overall algorithm performance. In fact, the sum of the three factors is several orders of magnitude smaller than the kernels execution time, which means a minimum overhead.

IV. Dynamic Scheduling

Task selection between all the enabled tasks is a key factor in the execution model. In the model implementation, Patterns from Object Oriented Design was chosen as a design tool. The design has three basic objects: one Petri Net, many Processors that interact with the PN, and Selectors that colaborate with the Processors, to select the next executable task.

Each Processor has a link to the Petri Net Object and a link to one Selector. The Selector object is responsible for defining the task that the Processor will do. It is a method that, taking as input parameters the state of the PN and the processor, returns the next task to the Processor. The design principle is to decouple processing from selecting tasks.

Different selecting policies are implemented by simply implementing the selecting method of Selector accordingly. This is a way to modelize homogeneous or even heterogeneous processors with different scheduling policies. Also, static or dynamic scheduling can be easily implemented using the appropriate Selector collaborator.

Simulation tests were developed to run static and dynamic tasks schedulers. In both cases, task assignation to a processor is dynamic, i.e. static or dynamic refers to the execution sequence, not the execution processor.

Two static schedulers are tested, following LL and RL algorithms. They were implemented easily by defining the order of tasks that Selector must follow. The sequence was defined from the algorithms shown in Fig. 1 and Fig. 2.

Two dynamic schedulers are tested. Both are based on DAGs, but differ in the selection metric applied. The first, called *height tree (HT)*, selects the enabled task that is higher in the dependency tree. The second, called *inverse tree (IT)*, select the enabled task that has a longer path to finish in the graph. By longer path we mean that it has a bigger number of steps in the longest path from the current to the end task. Non deterministic selection is done in case of equal height.

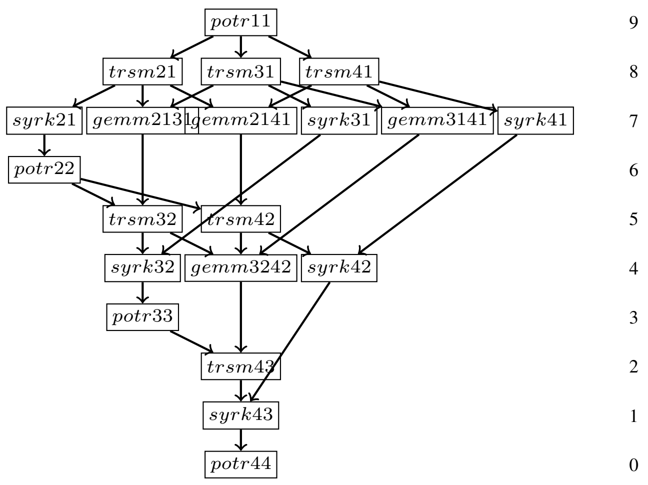

Fig. 6. Dependency graph of Cholesky algorithm, 4×4 tiles. Right values references to the stage number in which the task is enabled to fire (bigger value is earlier).

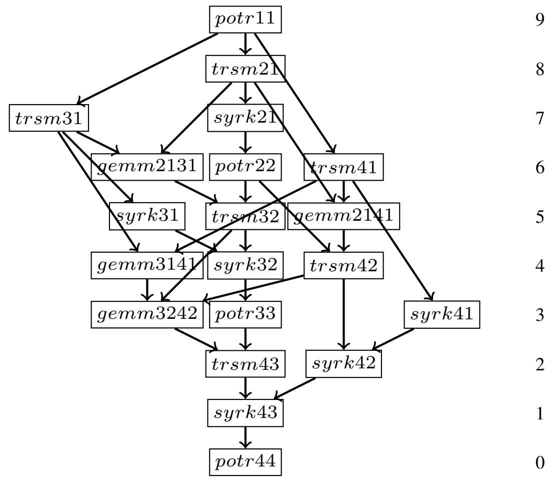

Fig. 7. Dependency graph of Cholesky algorithm, 4×4 tiles. Right values references to the latest stage number in which a task must to be fired (bigger value is earlier).

Figures 6 and 7 show examples of DAGs of dynamic schedulers used in tests. In *height tree* the level of a task is assigned according the step in which the task is enabled. In *inverse tree*, the level is assigned according to number of steps in the longest path to the end. For example, task *trsm41* has level 8 in the first graph and level 6 in the second.

Differences between both schedulers are exposed in the next example. Suppose you have three processors. Following the dynamic schedulers previously shown, the first step is compute *potr11*, and then compute *trsm21, trsm31* and *trsm41*. In the third step, scheduler *height tree*, must select any task from all those that have level 7 assigned, but scheduler *inverse tree* will select exactly *syrk21, gemm2131* and *gemm2141*. Is easy to see in Fig. 7 that *syrk21* is a priority tasks in the path to the end because it enables *potr22*. Scheduler *height tree*, due to its non determination, may delay it selection.

V. Simulation Results

Simulations of parallel algorithm were tested with different values for four parameters: matrix size, number of processors, number of block division and scheduler used. To test performance, a simulation tool was developed using a high level language (Smalltalk). The tool takes the number of blocks and defines all the tasks to be executed; then take matrix size and decides the block size, and finally it takes the number of processors and creates the same number of Processor objects and one thread for each of them to execute in parallel. According the scheduler, the respective Selector object is linked to each Processor. The execution of each kernel is simulated by throwing a time delay according the task.

In order to run simulations, it was used the time of running of each kernel, obtained for different block sizes, single precision, from a NVIDIA GTX 470 GPU. CUDA was used for BLAS kernels and MAGMA for LAPACK kernel xPOTRF. The results are shown in Table I. In all cases, the time of communication of all data from main memory to GPU and vice versa is considered. It was assumed that the main processor uses one thread to control each GPGPU.

Table II shows results of simulations with only four processors and block range of 6000 and 8000, single precision. Due to space limitations, only these results are presented, but they are representative of other combination of execution parameters. The metric of performance used is the idleness of processors, which is calculated as a difference between total execution time and total of processing time.

For Cholesky factorization algorithm, RL algorithm brings the best results for static scheduling, which are consistent with previous work [9]. For dynamic scheduling, *inverse tree* brings results which are near the optimum. A timeline for both schedulers is shown in Figures 8 and 9. Two things are noted: the idle time of processors in synchronization points in RL and the practical absence of idle time in IT.

Kernel	**Single pres. 6000**	**Double pres. 6000**	**Single pres. 8000**	**Double pres. 8000**
potr	0.249	0.882	0.509	1.895
trsm	0.568	2.018	1.122	N/A
syrk	0.465	1.907	1.001	N/A
gemm	0.755	3.506	1.678	N/A

TABLE I. Observed time for the kernels executed over an NVIDIA GTX 470 GPU, in seconds.

Block Size	**# Blocks**	**# Procs.**	**Algor.**	**Time (sec)**	**% idle time**
6000	6	4	LL	17.50	53.66
6000	6	4	RL	13.26	38.77
6000	6	4	HT	10.16	20.56
6000	6	4	IT	9.51	14.97
6000	8	4	LL	40.59	53.45
6000	8	4	RL	26.33	28.47
6000	8	4	HT	20.98	10.95
6000	8	4	IT	20.69	9.65
8000	6	4	LL	36.89	53.35
8000	6	4	RL	27.79	38.08
8000	6	4	HT	21.37	19.64
8000	6	4	IT	19.95	13.96
8000	8	4	LL	85.34	53.16
8000	8	4	RL	55.05	25.37
8000	8	4	HT	44.30	10.21
8000	8	4	IT	43.71	8.97

TABLE II. Observed values of simulations.

The nature of the Cholesky algorithm imposes no parallelism in the beginning and in the end of the execution, which sum four serial tasks. Beyond those points, and also at the end of execution, there is a limited number of parallel tasks which produce idle state for some processors. For the rest of the execution, all processors are always working.

VI. Conclusion and Future Research

We have developed a model of parallel programming starting from CPN, unfolding them to TPN and executed by a set of distributed processors that share in a memory area the representation of the state of the algorithm, and decouple the execution from the selection of the next task to do.

The model was used as a simulation tool, but it is easy to adapt it to running real algorithms. We hope to get performance improvements, due to the minimal overhead of the scheduling policy and its almost optimal "idleness" rate of processors.

The simulations were based on times taken from a currently usual multicore - multiGPU machine. Its results show that important improvements in performance can be obtained with respect to static scheduler algorithms, using a dynamic scheduler based on Petri Nets, which is easy to implement.

The model is adaptable to different numbers of processors and data block partitions: the unfolding of the CPN capture the number of partitions by generating the respective incidence matrix. Data dependencies are automatically generated. Besides, the execution model only needs as parameter the matricial information; thus, to execute different algorithms, no programming is necessary, it is enough to change the matrix.

Dynamic scheduling can be executed without need of previous time execution of each kernel. That information is necessary to achieve an optimal scheduling, at the cost of more complex schedulers. Our simulations show a result that is near the optimal, with a very light overload. Others dynamic scheduling policies may achieve optimal or sub-optimal results, but they are complex to understand and implement.

Execution on a set of asymmetric processors can be implemented by changing the Selector of task in each processor.

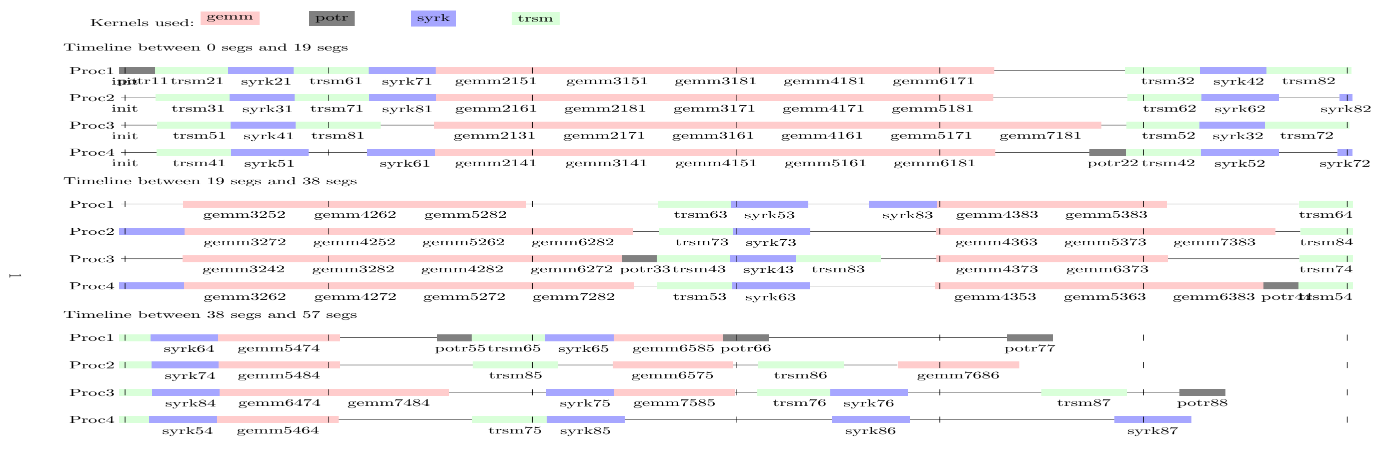
Fig. 8. Simulation timeline, RL scheduler, 8 blocks, 8000 range each, 4 processors

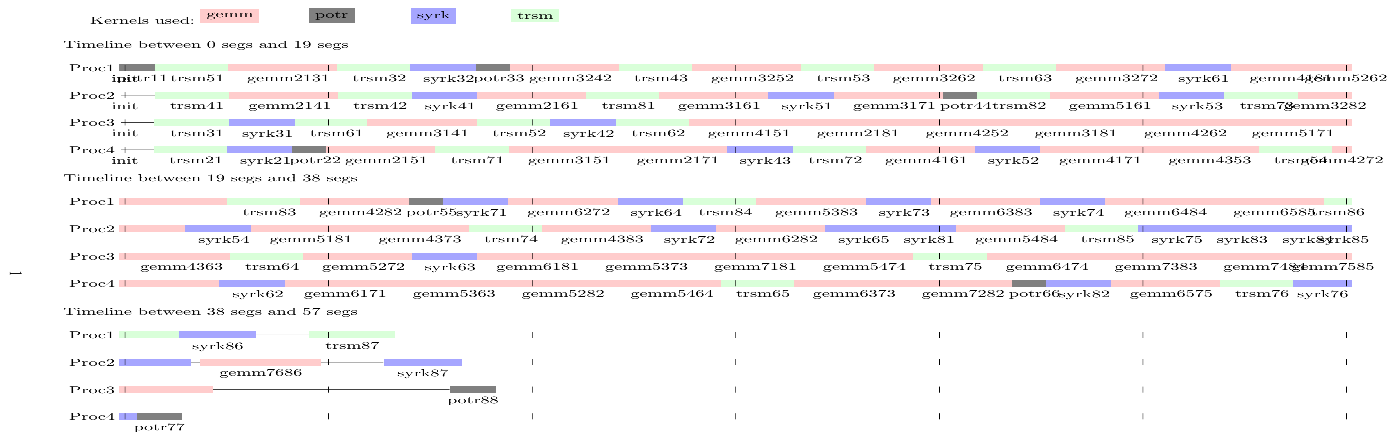
Fig. 9. Simulation timeline, IT scheduler, 8 blocks, 8000 range each, 4 processors

By restricting the execution of tasks that have forwarding dependencies in a non critical path to slower processors, those processors can help in the overall parallel execution.

Future work will implement the effective execution with this model, not only for linear algebra factorizations, but for others algorithms as well. An implementation in a distributed memory parallel architecture will also be researched.

Acknowledgment

The authors would like to thank to Profesor Orlando Micolini for his continuos sugestions and contributions.

References

[1] J. Kurzak and J. J. Dongarra, "Implementing linear algebra routines on multi-core processors with pipelining and a look ahead," LAPACK Working Note, Tech. Rep. 178, Sep. 2006.

[2] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," LAPACK Working Note, Tech. Rep. 191, Sep. 2007.

[3] "Basic Linear Algebra Subprograms Technical Forum Standard," University of Tennessee, Tech. Rep., 2001. [Online]. Available: http://www.netlib.org/blas/

[4] 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.)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999.

[5] J. Kurzak, A. Buttari, and J. J. Dongarra, "Solving systems of linear equations on the CELL processor using cholesky factorization," LAPACK Working Note, Tech. Rep. 184, May 2007.

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

[7] J. Hogg, "A dag-based parallel cholesky factorization for multicore systems," Technical Report RAL-TR-2008-029, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, Tech. Rep., 2008.

[8] J. Kurzak, P. Luszczek, M. Faverge, and J. Dongarra, "Lu factorization with partial pivoting for a multi-cpu, multi-gpu shared memory system." LAPACK Working Note, Tech. Rep. 266, Apr. 2012.

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

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

[11] K. Jensen and L. M. Kristensen, *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.