Algorithm model and execution based on Petri Nets in an heterogeneous parallel computer

Gustavo Wolfmann[1] and Armando De Giusti[2]

[1] Laboratorio de Computación - Facultad Cs. Exactas Físicas y Naturales
Universidad Nacional de Córdoba - Córdoba - Argentina
gwolfmann@efn.uncor.edu
[2] III LIDI - Facultad de Informática
Universidad Nacional de La Plata - La Plata - Argentina
degiusti@lidi.info.unlp.edu.ar

Abstract. Multicore - MultiGPU systems are frequently used in supercomputers design. The heterogeneity between both types of processors is a source of problems for the parallel programming: disparity in processing throughput and memory availability. While some problems are faster executed in a GPGPU, when its data size exceeds the memory available, data partition must to be done in order to resolve, and become desirable to use both types of processors. In this paper we present a solution based on Petri Nets to model the algorithm and to guide the execution, balancing the load between the CPUs cores and GPGPUs. The matrix multiplication algorithm is used as testbed. Tests confirm the goodness of the model and highlight the difficulties to address the problem.

1 Introduction

Currently, supercomputers are frequently based on nodes with a mix of multicores and GPGPUs[1]. These last are emerged as processing devices with a higher performance . Besides, the GPGPUs are dependents on the main computer, thus, the CPU processors are idle while the GPGPUs are processing.

The use of both kind of processors in parallel has two main problems: different throughput and memory size. While the system clock in CPU is generally faster than the clock in GPGPUs, the throughput for tasks related with vectorial processing is better in GPGPU. The counterpart is that GPGPUs devices have its own memory to allocate data, which is one order of magnitude smaller than memory available in the main system. The data size problem is resolved by dividing data and processing partially in the device.

To use simultaneously all the processing units available in a computer efficiently, each kind of processor must to be settled with its own set of parameters of data size and tasks to be executed. Thus, an asynchronous model must to be used in order to optimize the execution.

Petri Nets (PN) are well know as a tool to model parallel execution due its natural way of represent concurrency. Nevertheless, limited research was done

in the past to use it as an execution model in parallel programming. In previous works [11, 12] we have developed a technique to model algorithms based on PN's.

The gap between the algorithm model based on PN and the parallel execution is covered by a new parallel model of execution. The model is founded in two concepts. First, algorithm representation into PN is based in the concept that routines and their parameters are represented by transitions and places respectively. Second, in order to execute, processors are added to the model as the execution units. Simple linear algebra operations over the matrix representation of the PN model guide the execution.

A framework based on the model execution described before was developed and it is used as execution framework. The challenge is to model and execute an algorithm in an heterogeneous machine with parameters that allow to define execution options. This paper presents the results of tests done in an machine with two types of processors, multiples CPUs cores and two GPGPUs. By setting the parameters, as the division of data, the type and number of processors and the tasks associated with each one, several parallel configurations were executed, and confirms the goodness of the model.

The rest of the paper is organized as follows: the next section presents a brief summary of the Petri Net model. Section three introduces the execution model. The results and the conclusions are presented in the last sections.

2 The Petri Net Model

2.1 High Level Petri Net Model

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 [5]. This net is also known as Token Petri Net (TPN).

The Coloured Petri Net (CPN) is a type of Petri Net defined as ”High Level Petri Net“. The difference with TPN is that tokens have different values (”colors“) from a domain. This allows to model with a high level of abstraction. Here, transitions are enabled by having enough tokens in their input places respecting the ”color“ defined. The CPN definition is followed from [5, 8].

Petri Nets can be used to model algorithms, where operations (kernels to execute) are represented by transitions and data is represented by places. Input parameters are represented by arcs that go from places to transitions, and operations results, by arcs from transitions to places. The algorithm dependencies are implicit from the output arcs of transitions

Related to CPN, DAG's of task dependencies with many blocks divisions (tiled algorithms) are difficult to understand due to their large number of nodes (see Fig. 10 of LAWN 243 [7]). The CPN permit to model complex nets in a

simple manner. To model tiled algorithms [4], the main domain used to define colors is tile position, represented by the row-column pair.

The strategy to model a tiled algorithm is:

1. Each operation in the algorithm 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 data-block domain. Also, multisets are used to represent repetitions of blocks, and function arc expressions, to limit token flowing [8].

As the elected algorithm to test the model in the heterogeneous computer is Matrix Multiplication (MM), it is used as example to explain the CPN model strategy. Data division in MM is the key to execute in parallel, and there are many patterns to divide the data involved in the computation.

It will be considered three square matrixes of the same range, A, B, C, with $C = A \times B$. There are two frequently used ways to divide data, band division and tile division [4]. Band division divide A in n horizontal bands, B in n vertical bands, producing a division in C of $n \times n$ square blocks, such $C_{i,j} = A_i \times B_j$. Tile division divide all the matrix in $n \times n$ square blocks (tiles), such $C_{i,j} = \Sigma_k A_{i,k} \times B_{k,j}$. From the parallel execution point of view, tile division produces data dependency, as the final value of each block $C_{i,j}$ is the sum of n matrix with partial results. The kernel elected to execute is the xGEMM routine from BLAS library[2].

The fig. 1 has pictures of the banded data division, the CPN algorithm and its place domains. Transitions are named with the name of the routine represented. The name of the places follows the number of the block used in each operation. Multiset repetitions of tokens are represented as $\{x\}$. Fig. 2 shows the same three pictures for the tiled division. The domains are represented by a pair $< i, j >$ indicating row and column. There is an output link from transition *gemm* to place *gemm3* representing data dependency: the third parameter in its domain (q) indicate the number in the sequence of the partial computations.

2.2 The Executable Petri Net Model

In order to execute the algorithm, the overhead required to represent CPN domains and function arcs in an executable way is expensive in terms of high performance computing, and it is impractical to use it directly. Nevertheless, the CPN developed like this, meets the definition of well-formed CPN's [5]. They can be easily transformed into a TPN, which has a simpler computational implementation and is light to execute.

The unfolding of a CPN to a TPN is defined in Diaz et.al. [5]. Each place P_j in a CPN has an associated Domain $D(P_j)$; thus, its unfolding produces as many places in the TPN as the cardinality of $D(P_j)$, preserving the repetitions

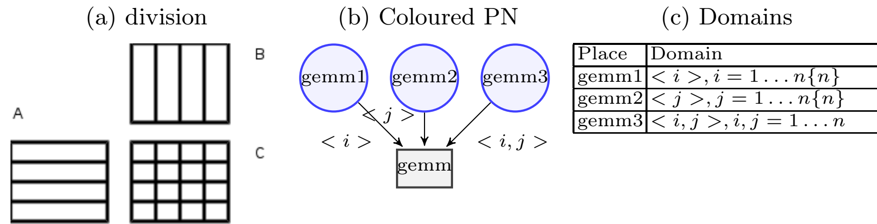
of the multiset. Hence, each place in the TPN has an association with a unique value from the pairs (color, place) in CPN and only one token can live on it.

Transitions are unfolded by generating as many transitions in TPN as the cardinality of the Cartesian Product of all the elements of its domain in the CPN. The cardinality of the multiset in each place must to be preserved. Hence, each transition in TPN is associated with a unique combination from the Cartesian Product of the domains of its input places, preserving multiset cardinality. Only guards with true values produce results. In this way, each unfolded transition represents an individual event, associated with a single task.

An unfolded example of the tiled division is depicted in fig. 3. It shows the series of tasks necessary to produce one resulting tile block of matrix C, under the assumption that the block is divided in 4 tiles. Each subtask mmi, $i = 1 \ldots 4$, has three input places, the respectives blocks of matrixes A, B and C. There is one output place in each task, to the third parameter of the next task, excepting in the last task, which produces the final result. This is due the data accumulation of partial results on C.

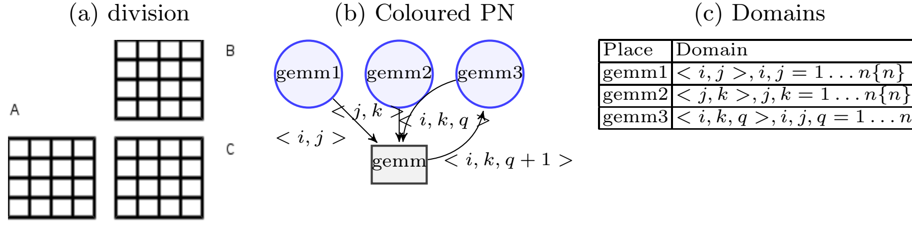
The unfolding exemplified must to be replicated for any tile block of matrix C, producing a big graph to be showed. Nevertheless, due each task is done only once, and each place is used for only one task, the TPN can be easily represented by a matrix notation, with incidence matrixes with only zeros and ones. These matrixes, jointly with the mark vector, form the elements to be used in the execution of the parallel algorithm, to be explained in next section.

Next, the model of the algorithm must to be adjusted to our heterogeneous environment. Both models of banded and tiled data division are introduced as a basis of the algorithm to be used in the heterogeneous system. The computer used in tests has four dies AMD opteron 6344 with with twelve cores each, 48 Gigabyte RAM, using only one memory channel, and a clock rate of 2400 Mhz. Each die has two blocks of L3 cache memory associated to a block of six cores. For floating point operations, the die has Fused Multiplication Addition (FMA) units shared by two cores. Each FMA performs concurrently one addition and one multiplication on vectors of 256 bits divided in records of 32 or 64 bits. Since there is an ACML version that is optimized for these FMA units, it was used, restricting the number of processors units to 24. As well as, the computer has



Place	Domain
gemm1	$< i >, i = 1 \ldots n\{n\}$
gemm2	$< j >, j = 1 \ldots n\{n\}$
gemm3	$< i, j >, i, j = 1 \ldots n$

Fig. 1: Banded division: Graphic, Coloured Petri Net and Domains of the Places.



Place	Domain
gemm1	$< i, j >, i, j = 1 \ldots n\{n\}$
gemm2	$< j, k >, j, k = 1 \ldots n\{n\}$
gemm3	$< i, k, q >, i, j, q = 1 \ldots n$

Fig. 2: Tiled division: Graphic, Coloured Petri Net and Domains of the Places.

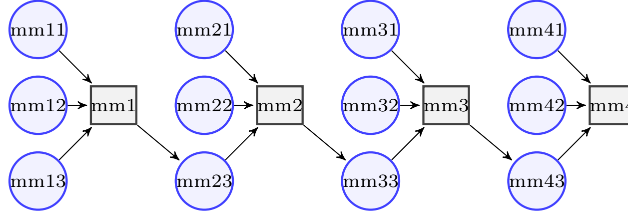


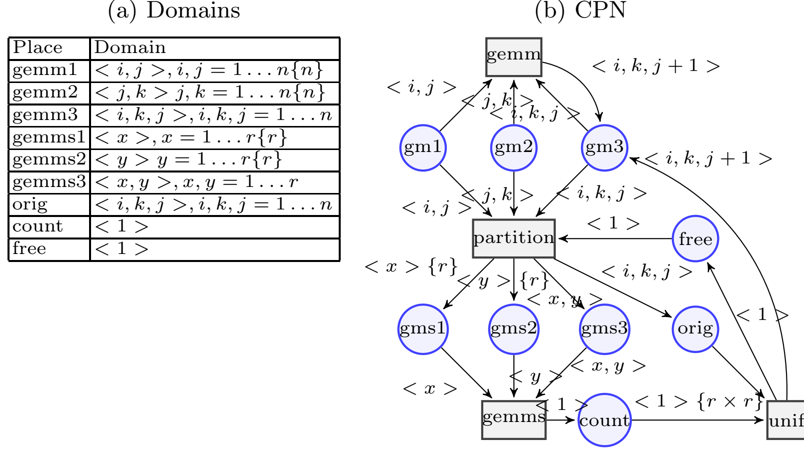
Fig. 3: Unrolled CPN of Fig. 2, for one resulting tile block, a generic $C_{i,j}$

two boards Nvidia GTX 680 mounted on the PCI bus of the board. Each one has 2 GByte of RAM and 1536 parallel threads of execution at a clock of 1006 MHz. Cublas library provide the kernel xGEMM.

To overcome the problem of heterogeneous processors, data blocks that produce the best performance on each kind of processors should be used. The different data size for each kind of processors requires dividing data in blocks of two sizes and synchronize the execution of each kind of processor. The solution is achieved by using both data divisions, one smaller to be used for the CPUs and one bigger to be used by the GPGPUs, and defining the smaller as a fraction of the bigger, in the idea that, CPU processors act as a set and can compute any bigger block by parts. The balance of load is based on this double division.

The double data division is done by a first tiled level and then a banded division over each tile. Thus, each tile is divided following the banded division. The double level division has the sense that, having a tile division of $n \times n$ blocks which defines n^3 tasks, each block can be computed by a GPGPU or by a set of cores of CPU. In the last case, if we define r as the number of band divisions for each tile, then, the complete tile is calculated by $r \times r$ bands multiplications.

Figure 4 shows the CPN that represents the double data division. The places that represent the three data blocks (*gemm1, gemm2 and gemm3*) are input places of two tasks: the *gemm* task, to be done by one GPGPU and the *partition* task, to be done by a core in CPU. This last subdivides each tile block in r bands, used by the *gemms* task (with a final ”s“ of subdivision). *Partition* is a virtual task because does not make any processing, only produces a set of logical



(a) Domains

Place	Domain
gemm1	$< i, j >, i, j = 1 \dots n\{n\}$
gemm2	$< j, k > j, k = 1 \dots n\{n\}$
gemm3	$< i, k, j >, i, k, j = 1 \dots n$
gemms1	$< x >, x = 1 \dots r\{r\}$
gemms2	$< y > y = 1 \dots r\{r\}$
gemms3	$< x, y >, x, y = 1 \dots r$
orig	$< i, k, j >, i, k, j = 1 \dots n$
count	$< 1 >$
free	$< 1 >$

(b) CPN

Fig. 4: Domains of the places for the CPN with tiled and banded divisions.

blocks, $< x >$, $< y >$, and $< x, y >$, used by *gemms* task. In order to avoid to do a *partition* of a new tiled block while there are not finished all the gemms tasks produced in a previous one, a control place *free* is added as an input to the *partition* transition which also acts as output place of *unification* transition. This last determines the completion of all *gemms* pending task, enabling a new *partition*, if necessary. In this way, the model to execute in the heterogeneous system is completed with a run time balancing. The next section describes the parallel execution model used to run in parallel the algorithm.

3 The Execution Model

The previous section shows how to model the algorithm with Coloured Petri Nets (CPN). Unfolding the CPN to a simple Token/Place Petri Net (TPN) transform a compact net into a bigger but simpler one to execute. This section shows how to execute a parallel algorithm based on TPN.

The Parallel Execution Model (PEM) is defined as a tuple:

$$PEM = (P, T, I^-, I^+, M, M_f, \Pi, \chi) \tag{1}$$

where P is a finite set of places P_i, with cardinality $|P| = p$, $i = 1 \dots p$; T is a finite set of transitions T_j, with cardinality $|T| = t$, $j = 1 \dots t$; I^- and I^+ are the negative and positive incidence matrixes of the TPN, with dimension $p \times t$ (I^- and $I^+ \in \mathbb{N}^{p \times t}$); M is the Mark Vector for places, $p \times 1$ ($M \in \mathbb{N}^p$); M_f, is the Final Mark Vector, $p \times 1$ ($M_f \in \mathbb{N}^p$); Π is a finite set of Processors Π_i, with cardinality $|\Pi| = \pi, i = 1 \dots \pi$ and each Π_i has a boolean variable e ($\Pi_i.e$), which is set as either true or false to indicate if it is running or if it is idle; and finally, χ is a Boolean variable that implements a mutual exclusion mechanism

over M that allows each Π_i to update M securely. The initial state of the net has $M = M_0$, the initial mark of the TPN; $\chi = true$, the exclusion is free; and $\Pi_i.e = true$, $i = 1 \ldots \pi$, because all processors are idle.

The PEM is very close to Timed Petri Nets [10]. Both share the concept that firing a transition is not instantaneous because there is a time elapsed between the start and the end of the firing. The same as in PEM, the firing action represents the execution of a task, but the difference is that in PEM firing is not done autonomously once the transition is enabled. An idle processor is responsible to fire the transition selected among all the enabled ones.

The implementation of this execution model needs one Mutual Exclusion (mutex) mechanism to avoid concurrent reading and writing operations over vector M, which is the one that defines the algorithm state. In this sense, the processors act serially to select the next transition to fire.

The Pseudo-code of the PEM execution is presented in Fig. 5. In round-robin format, each idle processor, searches for a task to execute based on the Petri Net model of the algorithm. To determine which transitions are enabled, only simple linear algebra operations are needed. In effect, if we call I_j^- and I_j^+ the j-th column (transition) in I^- and I^+ respectively, the j-transition is enabled if the vectorial subtraction $M - I_j^-$ does not have any negative value. Computing the vectorial subtraction for all the columns determines all the enabled transitions ready to be fired at one point of the execution.

To determine the task to be executed, a dynamic scheduler was developed. Each processor uses in run-time a valuation function that is applied to the set of enabled transitions, selecting the transition with highest valuation, T_k. The valuation function is the key for the parallel processing performance, because it can be particular to each type of processor in order to select to most appropriate task. By example, the faster processors can select tasks that will keep the larger number of tasks enabled in parallel, avoiding "bottlenecks" in the execution, and the slower processors select non-priority tasks.

Additionally, it is used a mapping between transitions and tasks to be executed, and another mapping between places and data blocks. To execute a task,

```
1  While main algorithm not finished
2    If can hold the mutual exclusion
3       Compute h function
4       Select one task to execute
5       Update M by absorbing tokens
6       Free the exclusion
7       Task execution
8       Inject tokens in M
9    Else
10      Delay
11   Endif
12 End
```

Fig. 5: Pseudo-code of the task selection algorithm.

the processor selects the task related to the transition to be fired and its data parameters from the first and second mapping respectively.

Steps 5 and 8 of the pseudo-code algorithm represents the evolution of the execution. The tokens are absorbed and injected from the Mark Vector M at two times. In step 5 the tokens from the input places of T_k are absorbed, and in step 8, they are injected to their output places. Both steps are made with linear algebra operations and, after injection, new transitions become enabled. The cycle is repeated until the end of the algorithm, which occurs when $M = M_f$.

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

The settings of mappings between transitions and places with routines and data blocks must not to be unique for all the processors, and is the key to adapt the model to an heterogeneous system. In our case, as there are two types of processors, each type has it own mapping settings, CPUs with ACML routines and small data blocks, and GPGPUs with CUBLAS routines and bigger data blocks. There is no need to synchronize both types of processors.

4 Experiments

The experiments were done on the multicore - multiGPU machine cited before, using gcc 4.7.2 as compiler and ACML 5.0 and CUBLAS 6.0 as BLAS implementation for the CPUs and the GPGPUs, respectively. The ACML version used was the tuned for use the FMA units in sequential mode based on two reasons: to get many logical processors to test the model and the poor speedup of the ACML parallel implementation. Thread affinity is used in order to assign the FMA units to the logical processors. A numerical single precision was used. The ranges of matrixes tested are 24000, 36000 and 48000.

The table in fig. 6a shows the results of use only one or two the GPGPUs, reaching more than 2 Tflops when the number of tiles is the small possible that allow fit data into the board. The results using only the FMA4 units are shown in fig. 6b, with a peak performance of 400 Gflops. The table in fig. 7 shows the results using both type of processors. The results of these test are not expected in the sense that the performance obtained is smaller than using only GPGPUs, with a best performance of 1.7 Tflops. The explanation for this is based on the bandwidth limitations of the memory channel and the PCI 2.0 bus in the motherboard. This conclusion is based on the irregular execution time observed in the execution of similar tasks when the data block is big, due to the channel saturation. Only the most meaningful results are shown in the tables.

Aditionally, the fig. 8 shows a timeline execution when two GPGPUs (light gray) and 16 cores or FMA4 units (dark gray) are running in parallel. The white

(a) GPGPU

tiles n		$n=2$		$n=3$		$n=4$		$n=5$	
ran	gps	secs	gflps	secs	gflps	secs	gflps	secs	gflps
24K	1	23.0	1202	26.7	1035				
24K	2	13.7	2018	15.5	1784	18.1	1527		
36K	1			76.3	1223				
36K	2			39.8	2344	49.2	1897	52.8	1767
48K	2					96.2	2299	114.8	1927

(b) CPU

bands r	$r=1$		$r=2$		$r=3$	
ran	secs	gflops	secs	gflops	secs	gflops
6K	1.51	286.1	2.20	196.3	2.29	188.6
12K	8.6	401.8	9.9	349.1	11.9	290.4
24K	77.7	355.6	64.8	426.7	88.0	312.2
36K			430.9	216.5	515.9	180.9

Fig. 6: Time in seconds and gflops from tests with several matrix ranges and divisions, two NVIDIA GTX 680 GPU's and 16 cores using FMA4.

		tiles/bnds		$r=3$		$r=4$		$r=5$		$r=6$	
range	gpus	cpus	n	secs	gflops	secs	gflops	secs	gflops	secs	gflops
24000	2	16	2	23.8	1161.6	19.3	1434.7	19.7	1403.4	20.8	1329.2
24000	2	16	3			24.4	1133.1	24.0	1152.0		
36000	2	16	2	136.5	683.6	160.3	582.1	95.7	975.0		
36000	2	16	3	60.7	1537.2	87.5	1066.4	54.7	1705.8	52.6	1774.0
36000	2	16	4	64.5	1446.7	74.0	1261.0			82.7	1128.3
48000	2	16	3			139.1	1590.1	188.6	1170.9		
48000	2	16	4	139.7	1583.3	195.1	1133.7	165.1	1339.7	306.3	721.4
48000	2	16	5			213.6	1035.5	269.6	820.4		

Fig. 7: Time in seconds and gflops from tests with matrix ranges of 24000, 36000 and 48000; differents tiles and bands divisions, two NVIDIA GTX 680 GPU's and 16 logical processors using FMA4 units.

spaces represent idleness of the processors. It can be shown that the execution is good in terms of activity, but not optimal, with white areas between the CPU processors, which is one of the reasons of the overall fall performance.

5 Related Works

The dynamic scheduler developed is related to Quark [13], but the our, in place of to prioritize the data locality as Quark does, it prioritizes the availability of parallel tasks.

StarPU is a runtime system developed at the INRIA institute that launches tasks in parallel over a set of processors units, using a dynamic scheduler [3]. To

Fig. 8: Execution timeline, 24000 range, 16 processors, $n = 2, r = 4$

run the tasks, it uses kernels provided by the user that implement the solution appropriate to each processor type. The scheduler uses estimated execution time to select the task to execute. The system is based on a library with routines that allows to define tasks, dependencies and data partition. These definitions must to be coded in the source program. It allows change the granularity of data in runtime and even use a granularity different for each type of processor, left to the programmer definition. Changes in any of these involve changes in the code, which is an important drawback.

XKaapi is another runtime system developed at Inria that launches tasks in parallel [6], with a different approach: it works based on compiler directives introduced in the source code that defines the tasks to run in parallel, like it does OpenMP. The scheduler is dynamic following a FIFO order without considering any other factor of optimization. Dependencies are automaticaly computed by the system. As in StarPU, changes in data division implicates to change the code. Besides, the task selection criteria between the set of available tasks is common to all the processors.

Shetti et.al. implements the HEFT (Heterogenous Earliest-Finish-Time) scheduling algorithm in a CPU-GPU environment [9]. They obtain an almost optimal performance on random DAG's. This scheduler is static and it is based on task priorities to select between them. It has the drawback to assign tasks priorities before running, which is a problem when having hundreds or thousands of tasks.

6 Conclusions

We emphasize the advantages to implement the algorithm with the developed model. First, as the number of data divisions is a parameter, the number of subtasks and its dependencies derives from the structure of the PN, making easier the analysis of the complexity and the performance of the parallel execution. Second, the unrolling process that transform the CPN to a TPN, hides the complexity of the parallel execution. Third, the parallel execution framework is configurable in order to execute differents algorithms in differents parallel machines. By doing changes in the matrix representation of the net, a new parallel algorithm can be executed without programming. Also, by changing the processors definitions, the execution can be ported to others parallel machines. Compared with related systems, ours is more flexible and quickly adaptable to changes. Few parallel systems can handle different data granularity according to the best to each processor.

The objective of the paper was achieved, due that the developed model execute the algorithm in both type of processors, CPU's and GPGPU's, with a convergent result. This paper, jointly with the ones done previously [11, 12], prove that the model can be used for differents algorithms and machines. The model helps to analyze, execute, and tune the execution, with a negligible overload.

Regrettably, an additive performance between both type of processors could not be achieved, due the hardware configuration and the parallel structure of the algorithm. Nevertheless, this was due on the extreme use of the processing resources. An important conclusion is that the bottleneck is produced by limitations in the memory channels and PCI bus, and that these factors must to be taken into account when a high performance is desirable.

The tuning of the execution parameters to reach better performance and the optimal execution on CPUs is left to next research, as also extending the model to a distributed memory machine.

References

1. Top 500 supercomputer site. http://www.top500.org/
2. Basic Linear Algebra Subprograms Technical Forum Standard. Tech. rep., University of Tennessee (2001), http://www.netlib.org/blas/
3. Augonnet, C., Thibault, S., Namyst, R.: Starpu: a runtime system for scheduling tasks over accelerator-based multicore machines. Technical Report 7240, INRIA (Mar 2010), http://hal.inria.fr/inria-00467677
4. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Tech. Rep. 191, LAWN (Sep 2007)

5. Diaz, M.: Petri Nets: Fundamental Models, Verification and Applications. ISTE Ltd - John Wiley & Sons, Inc., London, Hoboken (2009)
6. Gautier, T., Ferreira Lima, J.V., Maillard, N., Raffin, B.: 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), http://hal.inria.fr/hal-00799904
7. Haidar, A., Ltaief, H., YarKhan, A., Dongarra, J.: Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. Tech. Rep. 243, LAPACK Working Note (Mar 2011)
8. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer (2009)
9. Shetti, K.R., Fahmy, S.A., Bretschneider, T.: 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. pp. 212–218. PDCAT '13, IEEE Computer Society, Washington, DC, USA (2013)
10. Wang, J.: Timed Petri Nets: Theory and Application. The International Series on Discrete Event Dynamic Systems, Springer US (1998)
11. Wolfmann, G., DeGiusti, A.: Parallel asynchronous modelization and execution of cholesky algorithm using petri nets. In: Proc. Int. Conf. on Parallel and Distributed Processing Techn. and Appl. (PDPTA). Las Vegas, Nevada, USA (2013)
12. Wolfmann, G., DeGiusti, A.: Petri net based algorithm modelization and parallel execution on symmetric multiprocessors. In: Proc. Int. Conf. on Parallel and Distributed Processing Techn. and Appl. (PDPTA). Las Vegas, Nevada, USA (2014)
13. YarKhan, A., Kurzak, J., Dongarra, J.: Quark users' guide: Queueing and runtime for kernels. Tech. rep., Innovative Computing Laboratory, University of Tennessee (2011)