Petri Net Based Algorithm Modelization and Parallel Execution on Symmetric Multiprocessors

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 2014 - The Symmetric Multiprocessors architecture is composed by a complex set of cores, chips and memory channels that make it difficult to implement a parallel program that efficiently uses all resources. Another obstacle for achieving a performance according the resources is added by algorithms with hard data dependency. Asynchronicity is a key to get all processors running. Petri Nets have been used for a long time to model algorithms, but not as a tool to parallel execution. In this paper we introduce an asynchronous Parallel Execution Model based on Petri Nets and the process to go from a high level model to an executable parallel program. The Cholesky Factorization algorithm is used as a testbed. Tests results yield values that are near the theoretical peak and open good prospects to expand the model to other environments and algorithms.**

***Keywords*—Petri Net Modelization - Symmetric Multiprocessor - Parallel Execution Model - Cholesky Factorization Algorithm.**

I. Introduction

Tiled algorithms emerge as a solution to the problem of load balance for dense linear algebra algorithms on multicore processors [1]. This type of algorithms divides data in square blocks that are used in subsequent stages of processing until the final result is reached. This strategy of data division allows increasing the number of tasks that can be run in parallel when there is no data dependency among data tiles.

To improve processing performance, the parallel algorithm has to drive two key concepts: granularity and asynchronicity. Granularity plays an important role in parallel performance because the larger the number of blocks, the more tasks that can be run in parallel. Furthermore, without asynchronicity, performance decreases due the existence of blocking points in the algorithm that causes processors to become idle until they all reach each point.

The combination of a large number of tasks with asynchronicity generates an execution problem: the selection of the task to be launched and the processor that execute it. Hundreds or even thousands of tasks running in a machine with a large number of parallel processors will result in an overload problem.

In a previous work, [2] we have shown that Petri Net is a good tool to model and control the execution of a complex parallel algorithm. A high-level modelization based on Coloured Petri Nets (CPN) [3] has been presented. Transitions represent the tasks of the algorithm and Places represent the data parameters used by each task. For example, the Cholesky Factorization algorithm can be resolved with four BLAS/LAPACK [4], [5] routines; thus, the CPN that model that algorithm uses only four transitions with one, two or three input places according the data blocks used as parameters. No additional transitions nor places are needed.

The Coloured Petri Net model is good to understand and analyze the parallel algorithm, but it is too complex to be the basis for parallel execution. Unfolding a CPN to a Token Petri Net (TPN) [6] allows working with an equivalent but simpler net. This unfolding defines a net with the exact number of tasks the algorithm must execute. In addition, as Petri Nets are good to model a parallel process, the resulting unfolded net allows analyzing parallel execution restrictions of the algorithm.

On the other hand, there is a tendency to increase the number of cores in a Symmetric Multiprocessor machine (SMP). This feature is given by assembling multiple CPU chips into the motherboard, normally with two or four slots to add up to sixteen cores in each one. In order to maintain data locality, two problems arise with this hardware architecture: efficient memory management and process affinity [7].

The multiplicity of processors in the SMP machine results in difficulties with cache memory. Thus, the higher level of cache memory is shared by one or more cores, based on chip design. To avoid cache misses, a parallel process that uses a block of data, should not have more threads than the number of cores that share the higher cache memory.

Not only the number of threads must be present to minimize cache misses, but also the position of the thread in the pool of cores, namely, core affinity must be taken into account. If the threads of a parallel process are distributed in different chips of the CPU, cache consistence for all threads will have to copy data between chips, reducing performance.

Both facts, the number and placement of threads, have an impact on parallel algorithm design. In advance, a double core division is recommended to have all processors working with an acceptable level of cache faults. A first level consists in a series of logical processors, whose quantity must match the number of higher cache memory partitions. The second level divides each logical processor into as many threads as physical cores shares the higher level of cache memory. For example, if

the higher cache level is shared by four cores and the machine has 32 cores, it should be divided into eight parallel logical processors with four contiguous cores each.

Therefore, an efficient SMP machine utilization must not only use a data block size that minimizes cache misses, but also the number of cores that share one block of higher cache memory must be considered. If we add algorithm structure, the complexity to get an efficient parallel execution is high.

In this paper we present the design and execution results of a parallel version of Cholesky factorization algorithm, modeled with Petri Nets and executed on two different SMPs. This work is the continuation of the one cited above, using the same algorithm modelization. In this paper, an execution model that takes into account the hardware variables of an SMP machine is introduced. The rest of the paper is organized as follows: the next section presents a brief summary of the Petri Net model developed before. Section three introduces the execution model. Results are discussed in Section four and finally, conclusions and future research are presented.

II. The Petri Net Model

In a previous work [2], the model used to analyze and simulate runnings of a parallel algorithm was introduced. It is based on Coloured Petri Nets. This high level model is then unfolded into a Simple Place / Transition net (TPN), which is used to run the parallel algorithm. A summary is presented.

Figure 1 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 $< x,y >$, multiset repetitions by braces $\{x\}$, and functions arcs are only Booleans of the form $if(cond)$.

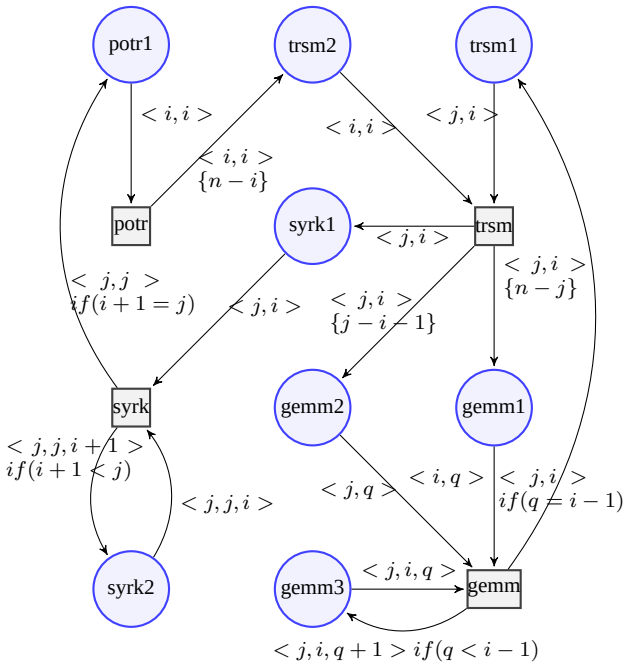


Fig. 1. Coloured Petri Net that represents Cholesky's factorization algorithm.

Place in CPN	Domain in CPN
potr1 trsm2	$< i,i >, i = 1 \dots n$
trsm1 syrk1 gemm1	$< j,i > j = 2 \dots n, i = 1 \dots j-1, j > i$
gemm2	$< j,i >, j = 3 \dots n, i = 1 \dots j-2, j > i$
syrk2	$< j,j,i >, j = 2 \dots n \wedge i = 1 \dots j-1 \wedge j > i$
gemm3	$< j,i,q >, j = 3 \dots n, i = 2 \dots n-1, q = 1 \dots i-1 \wedge j > i \wedge i > q$

Fig. 2. Domains of the Places for the Coloured Petri Net in Fig.1 .

Arcs domains are shown in the table of Fig. 2. The algorithm is generically defined by the CPN, and the number of tiles in which the matrix is divided is provided as a parameter in the domain definition. The high level of expressivity of a simple model can be remarked.

In contrast, 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 CPNs [6]. This type of nets is easily transformed into a TPN, which has a simpler computational implementation and is light to execute.

The unfolding of a CPN is defined in Diaz et.al. [6]. 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 $D(P_j)$ in the colored Place, 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.

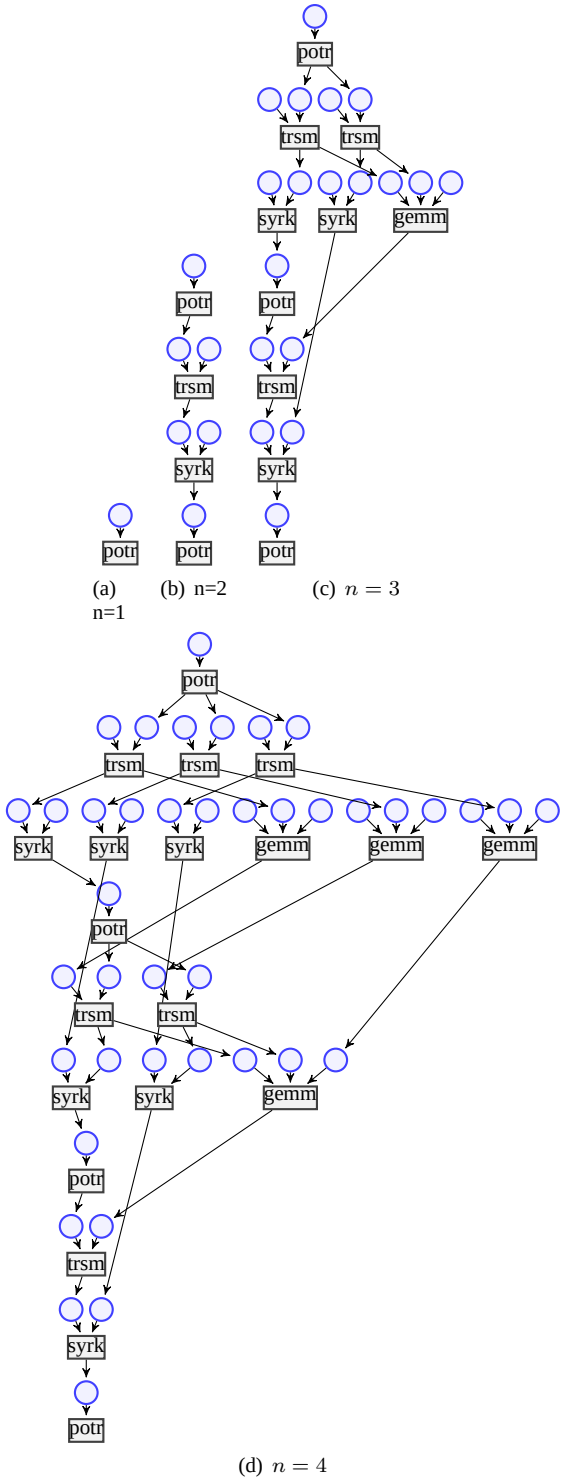
Transitions are unfolded by generating as many Transitions in TPN as the cardinality of the Cartesian Product of all the elements of its domain in the CPN. The cardinality of the multiset in each Place must be preserved. Hence, each Transition in TPN is associated with a unique combination from the Cartesian Product, preserving repetitions of the multisets of each input Place. Only guards with true values produce results. By construction, each unfolded Transition represents an individual event that will be associated with a single task.

Figure 3 shows four examples of unfoldings from CPN to TPN, for $n = 1, 2, 3, 4$ square tiles divisions[1]. Places are shown in the same order they have in CPN. Their names are not shown due to space limitations. The order in which each unfolding is shown is not random: each unfolding of n divisions has the same graphic as $n-1$ divisions, adding to the top, the tasks due to larger number of tiles. In this way, the Transitions in Fig. 3 at the same horizontal level represent the same task in all figures, ordered from end to start.

The chart in Fig. 3 shows two important aspects of algorithm parallel execution. First, there is a critical path of tasks execution derived of data dependency, that can not be exceeded [8]. Each increment in the number of tiles generates a sequence of *potr*, *trsm* and *syrk* tasks that must be done serially.

[1]A tile division of n represents $n \times n$ square blocks of data

Secondly, in the hypothetical case of having an unlimited number of parallel processors with the same execution time for all tasks, the minimum time required for the parallel execution is a function of the number of tile divisions, and clearly, there is an upper limit of the number of processors that can run in parallel.



(d) $n = 4$

Fig. 3. Token Petri Net unfolded from the Coloured Petri Net in Fig.1 using different numbers of square tiles divisions (n).

op \ 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. 4. Number of each task according the tile division.

Figure 3 shows that, when the number of tile divisions is small, the idleness of all parallel processors is high, due the strong data dependency. The table in Fig. 4 shows the number of algorithm tasks based on the number of tile divisions. The *potr* task has a linear growth, *syrk* and *trsm* have quadratic growth, and *gemm*, cubic growth. As stated above, each stage of processing introduces a group of three serial tasks, except for the first one has only one. This implies that, for example, a tile division of five has 13 sequential tasks over the total of 35.

The parallelism of the algorithm with few tile divisions is poor. Due the cubic growth of the *gemm* task, more divisions generate more tasks and a better capacity for parallel execution. By contrast, a tile division of 10 results in a large number of parallel tasks, and an increment of the overload cost. A tile division with low number of divisions in a SMP machine with few cores, will be enough to make an acceptable use of its computational power, but, in machines with 32 or more cores, a bigger number of divisions and a complex management tool of tasks to exploit their capacity will be needed.

In the next section, the parallel execution model is described. It was developed to handle the combined complexity of the algorithm and the SMP architecture, and it will be used as a basis for the execution.

III. 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:

$$PEM = (P, T, I^-, I^+, M, \Pi, \chi) \quad (1)$$

where:

- P is a finite set of Places P_i, with cardinality $|P| = p$, $i = 1 \ldots p$.
- T is a finite set of Transitions T_j, with cardinality $|T| = t$, $j = 1 \ldots t$.
- 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 $|\Pi| = \pi, i = 1 \ldots \pi$. Each processor 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.

- 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 is:

- $M = M_0$, the initial mark of the TPN.
- $\chi = true$, the exclusion is free.
- $\Pi_i.e = true$, $i = 1 \dots \pi$, because all processors are idle.

The PEM is very close to Timed Petri Nets [9]. 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 as it is in Timed Petri Nets. An idle Processor is responsible to fire the Transition selected among all the enabled ones.

The number of enabled Transitions can be lower or higher than the number of Processors. As a result of this, we can have idle Processors with no Transitions to fire or enabled Transitions waiting for a free Processor, depending on the number of enabled Transitions in relation to the number of idle Processors. In the first case, the execution speedup of the will be poor and this situation must be avoided. In the second case, the Processor must select the most appropriate Transition to fire. To do this, we have adopted the general criteria of selection based on the priority of the Transitions that are in the "Critical Path" to finish the algorithm.

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 when selecting the next Transition to fire, waiting for the mutex enabled.

The Pseudo-code of the PEM execution algorithm is presented in Fig. 5. In round-robin format, each logical processor with the enabled flag set on, searches for a task to execute based on the Petri Net modelization 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. This is defined by function h, which has arity $h : \mathbb{N}^{p \times t}, \mathbb{N}^{p \times 1} \to \{0, 1\}$, with parameters M and I^-, and their result values are:

$$h(j) = \begin{cases} 0 & \text{if } (M - I_j^-) \text{ has negative/s value/s} \\ 1 & \text{if } (M - I_j^-) \text{ else} \end{cases} \qquad j = 1 \dots t$$

Computing h for all the columns determines all the enabled transitions ready to be fired at one point of the execution. By design, each Place of the unfolded TPN is the input Place of only one Transition. This guarantees no competition between

```
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.

enabled transitions for input tokens, and that all enabled transitions can be fired simultaneously.

To determine the task to be executed, a dynamic scheduler was developed. It uses 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 when the set of enabled Transitions has more than one element, it must select the one that will enable more Transitions in the future, namely, it keeps the larger number of parallel tasks enabled. This scheduler is related to Quark [10], which prioritizes data locality instead of availability of parallel tasks.

Additionally, there is a mapping between each Transition and each task to be executed, and also a mapping between each Place and the data block which is used as parameter in the task. To execute a task, the Processor determines which task and which parameters are needed from the T_k selected and then runs it.

Steps 5 and 8 of the pseudo-code algorithm represents the evolution of the execution. Similar to Timed Petri Nets, tokens are absorbed and injected 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 simple linear algebra operations:

$$M' = M - I_k^- \qquad \text{in 5 at } t_0 \qquad (2a)$$

$$M''' = M'' + I_k^+ \qquad \text{in 8 at } t_0 + \Delta_k \qquad (2b)$$

and after step 8, potentially new Transitions become enabled. M and M'' are the markings at time t_0 and $t_0 + \Delta_k$, where Δ_k is the elapsed time of the T_k task execution. The cycle is repeated until the end of the algorithm is reached, 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 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.

IV. EXPERIMENTS

A FORTRAN program was developed to read, interpret and execute the model. OpenMP was used as shared memory model of execution and tests were run over two machines, the first with four AMD 6344 processors, which conform a 48 cores machine, and a second, with two Intel Xeon E5-2680 chips. Single precision floating point was used for all tests.

There are two requirements of the PEM that limit the possible stack of compiler / libraries to be used for coding: nested parallelism and core affinity. Both requirements are the basis of the PEM and are features that must be present jointly in the compiler. This fact left only one possibility for each machine: gfortran with ACML for the AMD-based machine, and Ifortran with MKL for the Intel-based one.

The configuration of the parallel hardware may vary in number of cores and chips, so the implementation of the PEM must be configurable to adapt to the hardware used. Thus, an XML-style file that contains the settings of the real machine in which the program runs is used. It contains the first-and second-level processor division, the type of physical processors and the mapping between Places with data and Transitions with tasks. In the software, this feature acts as an intermediate layer between hardware and algorithm and make tunning easier.

A. The AMD-based machine

The architecture of the AMD-based is as follows: four dies with twelve cores each. Each die has two blocks of L3 cache memory associated to a block of six cores. For floating point operations, the die has one Fused Multiplication Addition unit (FMA) shared by two cores. Each FMA unit can perform concurrently one addition and one multiplication of 256 bits, improving the processing power of the cores that share it. Since there is an ACML version that is optimized for these FMA units, it was used but restricting the number of processors to the model in the PEM model to 24.

The logical hardware division used in the tests was $n \times 1$ processors, where n is the number of first level processors, i.e., the second level of divisions has only one core. This decision was made based on two factors: the first is that the research focus is over task synchronization, thus, the higher the number of processors to synchronize, the better the test to our objectives. Second, the implementation of ACML for the routines used, has a poor speedup. In effect, the parallel implementation that uses the FMA units, when using several threads (six,eight,etc), does not scale properly for the routines *ssyrk*, *strsm* and *spotrf*. In consequence, the serial version of ACML was used, running each routine in one FMA unit. Nevertheless, the affinity concept remains important, because, as the FMA unit is shared by two cores, the logical processor division needs to take two consecutives cores to make exclusive use of one FMA.

Several test results are shown in Table 6. The body presents time and flops of various combinations of matrix range, number of parameters and tile divisions.

The analysis of the results brings some conclusions:

- The low number of data divisions has poor results: it is the effect of poor parallelism when the number of tile divisions is fewer than six.

procs		8		16		24	
range	dvs	secs	flops	secs	flops	secs	flops
12000	8	3.14	183	2.89	199	3.12	185
	12	2.91	198	2.21	260	2.38	242
	15	4.05	142	4.07	141	4.42	130
24000	8	22.11	208	18.98	243	20.33	227
	12	19.28	239	13.49	342	14.98	308
	15	19.21	240	12.06	382	13.73	336
30000	8	43.05	209	36.99	243	40.90	220
	12	36.02	250	25.11	358	25.74	350
	15	35.50	254	23.11	389	23.42	384

Fig. 6. Time in seconds and flops in GFlops from tests with matrix ranges of 12000, 24000 and 36000; 8, 16 and 24 processors, and data divided in 8, 12 and 15 tiles with the AMD-based machine.

- A tile division of 15 generates 680 tasks with 1800 parameters, which is the range of the corresponding incidence matrix. This results in a heavy overload for the matrix operations when updating the Marking Vector M. However, this tile division produces a large number of parallel tasks. A balance must be achieved between the number of parallel tasks and data tile range in order to keep the processing/overload ratio convenient for throughput.
- The best result is for range of 30000, 16 processors and 15 tiles, which brings 389 Gflops. Since AMD-based machine has a theoretical processing peak of 998 Gflops[2], it represents a processing utilization that is close to 40% of its peak. Better yet, if we only consider the sixteen processors used, the effective processing ratio increases to 58%. These are very good values considering they are negatively affected by cache misses, the serial part of the algorithm and overload.
- Using the 24-processor configuration has no speedup improvements versus using the 16-processor one. This is due to a problem in physical memory configuration because the bank one is the only one used in this machine, so the memory channel becomes saturated when all the 24 processors are running.

B. The Intel-based machine

The Intel-based machine used has two Xeon E5-2680 chips, each of them with eight cores. Thus, the operating system has a total of sixteen threads available.

Each of the cores of the Intel processor has one AVX unit (Advanced Vector Extensions) that uses 256 bits registers. It can perform addition and multiplication operations simultaneously over these registers. This feature determines a theoretical processing power per core similar to that of the FMA unit available in the AMD-machine.

In the Intel-based machine, tests were executed using Intel Composer 2013 suite, which includes the MKL BLAS/LAPACK implementation. In Fig. 7, a summary of the most representative results are shown. As with the AMD-based machine, the best performance is reached when tile division is twelve.

The analysis of the results brings some conclusions:

[2]998 Gflops = 2.6 Ghz x 24 fma units x 2 ops x 8 single precision values

procs		8x1		16x1	
range	dvs	secs	flops	secs	flops
24000	8	17.08	270	13.25	348
	12	14.35	321	9.52	484
48000	8	140.59	262	101.73	342
	12	109.24	337	70.99	519

Fig. 7. Time in seconds and flops in GFlops from tests with matrix ranges of 24000 and 48000, 8 and 16 processors, and data divided into 8 and 12 tiles, with the Intel-based machine.

- Similar to the results obtained with the AMD-based machine, the impact of having more tile divisions is strong: in all cases, division by 12 tiles increases performance goes up to 30%.
- Going from 8 to 16 logical processors does not scale properly due to the effect of memory channel saturation. However, performance goes up to 50%.
- Processing a big matrix with 16 logical processors and 12 tile divisions yields a result of 519 Gflop. Considering the processing power available with sixteen cores, the rate of use of the physical processors gets near 75% of the theoretical peak, which evidences a very good management of cache misses and synchronizations.

A final test was done by fixing the number of tile divisions and modifying core divisions. The table in Fig. 8 shows time and flops for a range of 24000 and 48000, 12 tile divisions and all the remaining combinations for sixteen cores into two levels. This test was done on the AMD-based machine, but the scalability was so poor that the results were useless. This is the opposite with the Intel-based machine. They show a similar performance regardless of how the cores are divided. The best performance is achieved with a division of four logical processors with four cores each, which brings 616 gflops. Considering that the theoretical peak of the machine is 691.2 gflops, a near-optimum result was obtained.

procs		1x16		2x8		4x4		8x2	
range	dvs	secs	flps	secs	flps	secs	flps	secs	flps
24000	12	11.69	392	10.17	453	8.98	513	8.40	549
48000	12	69.73	529	62.72	588	59.81	616	60.33	611

Fig. 8. Time in seconds and flops in GFlops for tests with matrix ranges 24000 and 48000, using 16 threads with double level division, virtual processors x internal threads (1x16,2x8,4x4,8x2), and data divided into 12 tiles, with the Intel-based machine.

Finally, the Fig. 9 shows a timeline for the execution of one of the tests, with a range of 24000, a division of 12 tiles and 16 processors. Processor idleness can be observed both at the initial and final stages of the execution, as mentioned above. Also, there is no idle time while processing and the low impact of the overload is evident from the absence of idle areas around the tasks.

V. Conclusions

The research has several points to highlight:

- It has been shown that Petri Net not only has good properties to model concurrent systems, but that it is also a good basis for a parallel execution model. It is not easy to manage the parallel execution of hundreds or even thousands of tasks, but we found how to do it with this tool.
- The execution tool developed can be adopted to any SMP machine and optimized libraries thanks to the combination of TPN with the virtual processor definition, two levels of hardware division and processor affinity.
- The parallel execution environment developed was able to reach a real utilization of the processors very close to its theoretical limit. Asynchronicity and affinity were the key to this achievement.
- Modeling an algorithm with CPN allows analyzing its parallel capabilities and brings information about its possibilities and limitations in the search for better parallel performance. In particular, we conclude that applying a tiled division of data to the Cholesky algorithm does not result in a good performance if tile division is less than six.
- The XML-style of the model's parameter file allows not only adapting it to different machines, but it also leaves open the capability to switch the algorithm by only changing the incidence matrix and task mapping. Thus, any parallel algorithm designed following a well formed CPN can be executed with a high level of performance by only changing the incidence matrix and tuning its virtual processors.

Future work will focus on researching the advantages and obstacles when using the technique developed with various algorithms and also using heterogeneous systems, such as hybrid CPU / GPU systems. An implementation in a distributed memory architecture will also be studied.

References

[1] 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.

[2] G. Wolfmann and A. De Giusti, "Parallel asynchronous modelization and execution of cholesky algorithm using petri nets," in *The 2013 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA13)*, 2013.

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

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

[5] 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.

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

[7] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012.

[8] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid gpu accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, 2010.

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

[10] A. YarKhan, J. Kurzak, and J. Dongarra, "Quark users' guide: Queueing and runtime for kernels," Innovative Computing Laboratory, University of Tennessee, Tech. Rep., 2011.

Fig. 9. Execution timeline, divided in two sections, Intel-based machine, 24000 range, 12 tiles, 16 processors