Parallel Execution on Heterogeneous Multiprocessors from Algorithm Models Based on Petri Nets

GUSTAVO WOLFMANN
Universidad Nacional de Córdoba
Facultad de Cs. Exactas, Físicas y Naturales - Lab. de Computación
Av Velez Sarsfield 1611, 5016 Córdoba
ARGENTINA

Abstract: Current supercomputers are composed by nodes containing a combination of general purpose computing units (CPUs) and specific mathematical coprocessors. In this way, GPGPUs or Xeon Phi cards are attached to the nodes to improve its performance. Both types of processors, CPUs and coprocessors, have many differences, like their architecture, the clock rate of the processors and the operation of the related memory. These are the main factors that conform an heterogeneous multiprocessor. A parallel program that wants to achieve the sum of the performance of both types of processors, must consider not only the complexity of the parallel algorithm, but also the differences in the architecture of processors, increasing its complexity. As a contribution on this problem, this paper presents a model of parallel execution based on Petri Nets, called PN-PEM, that is used not only to model a parallel algorithm, but also to execute it directly on a computer with heterogeneous multiprocessors. An asynchronous execution of tasks and a dynamic scheduler are the main characteristics that allow execute a parallel program on this type of parallel computer. Tests done on a multicore computer with two Xeon Phi cards reach the aggregate performance of both type of processors, confirming the quality of the model used.

Keywords: Heterogeneous Multiprocessor, Parallel Algorithm, Petri Nets, Asynchronous Parallel Execution, Dynamic Scheduler.

1 Introduction

In these days, supercomputers are composed by nodes which are a combination of symmetric multiprocessors (SMP) and coprocessors (or accelerators) used to improve numerical processing [13]. The tendency is to incorporate more and more this type of coprocessors specifically designed to this type of processing.

Without going into details of most used accelerators, they provide an unbeatable power of computing due to its specific architecture optimized to mathematical and logical processing. The key of the performance are hardware parallel threads that allow to run a high number of tasks in parallel. Also the memory layout in the card plays a rol in the performance. Althouth the clock rate is slower than the clock rate of CPUs, the existence of a large number threads provides a high performance.

Nevertheless, main processors have also a computing power that has not to be neglected. Mainly by the faster clock rate and by the specific hardware designed to execute instructions with 256 or 512 bits registers. Thus, the computing power available is a mix of the traditional CPUs and the new coprocessors cards, conforming an heterogeneous multiprocessor computer.

To run a parallel program that uses both type of processors requires of a scheduler that launches tasks on each type of processors properly. A synchronous execution between both types produces a loss of performance due to the load unbalance. An asynchronous execution is recommended in order to avoid idleness in the processors, with the drawback of increasing the parallel processing overload.

In a pair of previous papers [15, 16], it has been proved that Petri Nets are a good tool to model and control the execution of a complex parallel algorithm. It was introduced an algorithm modelization based on Colored Petri Nets (CPN) [9]. The strategy to model with CPNs is based on two concepts: each task of the algorithm is represented by a transition and its data parameters are represented by places linked to the transition that represents the corresponding task.

The execution of a parallel program based on CPN is complex due to its high level semantics. If some constraints are followed in the stages of modeling with CPN, the net obtained can be transformed into a simple Token Petri Net (TPN) model that preserves its semantics. The TPN model of the algorithm is used as a basis for the parallel execution.

A framework named PN-PEM, from Petri Net Parallel Execution Model, was developed to execute a parallel program from its TPN representation. It allows to configure the number and kind of processors to execute in parallel the algorithm, and also allows to assign to each processor, the routine used to execute each task. In other words, it

is needed to provide the kernels that implement each task of the algorithm and assign them to the respective hardware.Another point of configuration is the scheduling policy to launch tasks in each processor, which is not limited to be a single general policy, but a multiple and proper to each processor.

The configuration of the framework, by the means of the definition of processors, the kernels for each type of processor and the scheduling policy, allows to the user configure the parallel computer to obtain the best results of the execution. In the practice, if the parallel computer is symmetric, is needed only one kernel that implements each task. If it is heterogeneous, multiple kernels for each task are needed, according the kind of processors .

This paper presents an implementation of a parallel algorithm executed on an heterogeneous multiprocessors computer with Xeon Phi accelerators. The work is a continuation of the ones cited before, introducing an assignment of tasks in an heterogeneous multiprocessor. The main objective is to achieve a performance close to the sum of the individual performance each type of processor. The hypothesis is that the flexibility to adapt the scheduling policy according the type of processor will play an important role in the performance reached.

The algorithm used as testbed is the Cholesky factorization. This algorithm has an intermediate level of data dependency along its execution, and it has been widely studied with a large number of proposed solutions with high performance [3, 7].

The rest of the paper is organized as follows: next section presents a brief summary of the PN-PEM model developed before. Section three introduces the execution model of the framework. Tests and results are discussed in Section four and finally, related works, conclusions and future research are presented.

2 The Petri Net Model of the Parallel Algorithm

Directed Acyclic Graphs (DAG) have been used to model algorithms, with vertices representing the tasks and edges that represent the dependency among them. The resulting graph is also known as Dependency Graph. It helps the scheduler to follow the dependency between tasks when selecting the next to be executed [3].

The Petri Net model is a mathematical model that represents processes that have concurrency between them [8]. It is based in the graph theory, adding the notion of execution of the model. It conforms a theoretical model suitable to represent parallel algorithms as it is showed in a recent work [2]. Nevertheless, it exists a gap between the model and the execution of the algorithm, under which, it is not widely used.

The main focus of the framework PN-PEM is to take advantage of the modeling features of the Petri Nets while it solves the execution gap problem. Two guidelines were kept in its developing. First, preserve the semantics of the CPN model of the algorithm in its stage of execution, and second, give to the user the maximum flexibility to configure the parameters of parallel execution.

In the PN-PEM framework, Petri Nets (PN) are used to model the algorithm, based on two premises. First, transitions of the PN represent operations (kernels to execute), and second, the places represent the data involved in the operations. Input places of a transition represent the data parameters of the task, and output places represent its results. A token in an input place represents that the respective data are available to process. A transition is available when all its input places have tokens, thus, all data needed is present. The data dependency between tasks is implicit when a place acts as output of a transition and as input of other transition, namely, the result of a task is used as input of other task.

The asynchronous execution of the PN is represented through the out of order firing of transitions. Since transitions represent actions, the semantics of the firing is not the immediate execution as in the simple PN model, due to it involves a time lapse. This fact is well formalized by the model of Timed Petri Nets [14]. In the PN-PEM model, when a transition is fired, tokens are absorbed immediately from the input places and are injected into the output places once the time involved in the execution of the corresponding task is elapsed.

Colored Petri Nets (CPN) allow to model complex nets with high level of abstraction in a simple manner. In the Dependency Graphs, each task is represented by a vertex, which implies that, as it increases the number in which the overall algorithm is divided in order to execute it in parallel, it increases also the number of vertex in the graph, difficulting its understanding, as can be seen in the LAWN 243 [7]. The color in the tokens of the CPN allows to generalize the model without increasing the number of transitions and places.

The data division in this work follows the “Tiled Algorithms” of the LAWN 191 [3]. Thus, data is divided in a 2D pattern, naming each block by a row-column pair. Pairs are used to define the domain of the places of the CPN.

The Cholesky factorization was chosen as a testbed algorithm. The computations use the same kernels as in the LAWN 223 [10] for tiled algorithms. 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. The kernels provided to the framework are just the implementation of these routines in the respectives physical processors that execute them.

Figure 1 shows the CPN that represents Cholesky’s algorithm. It has only four transitions and eight places; each transition represents a task and each place represents one of its data parameters. The name of the transitions follows

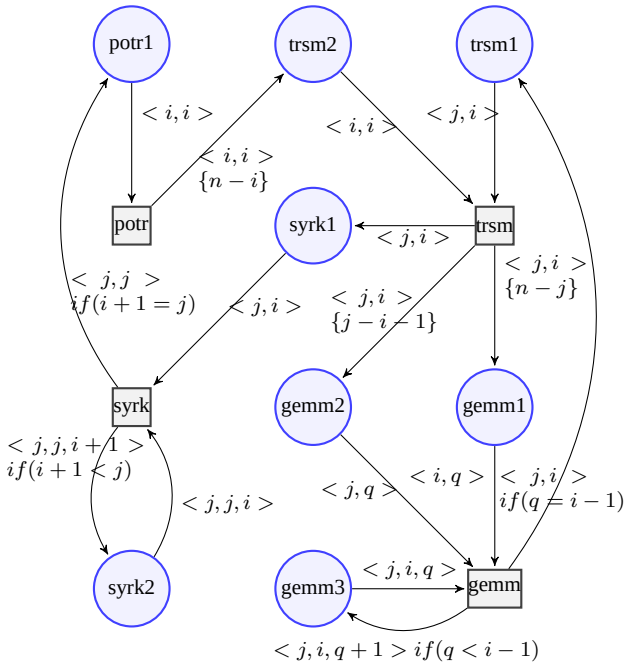

Figure 1: Colored Petri Net that represents Cholesky's factorization algorithm.

the name of the routine that it implements. The name of each place is the number of the block used in each operation, which follows the 2D data division of the matrix; then, colored tokens are represented by the pair $< x,y >$. CPNs use multisets, whose repetitions are represented by braces $\{x\}$. In addition, the arcs may have functions associated with them, which are restricted to be Booleans functions of the form $if(cond)$.

The domains of places are shown in Table 1, where $< n >$ is the parameter that represents the number of tile divisions. This last element completes the model of the algorithm.

The direct execution of the algorithm represented by the CPN is complex and expensive in terms of high performance computing, due to the high level semantics of the CPN. It would be necessary to implement a runtime interpreter that translates color domains into block positions in the matrix, and evaluates boolean expressions, which is costly in terms of performance.

Nevertheless, the CPN developed as described before meets the definition of well-formed CPNs [4]. This type of net can be transformed into a TPN, which has a simpler computational implementation.

The transformation of a CPN into a TPN is called "unfolding" and is defined in Diaz et.al. [4]. The unfolding takes each place P_j in the CPN and produces as many places in the TPN as the cardinality of its domain $D(P_j)$ in the CPN. Hence, each place in the TPN has an association with a unique value of its color.

Transitions are unfolded by generating as many transi-

Places	Domain
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	$< j,i,q >, j = 3...n, i = 2...n-1,$ $q = 1 \dots i-1 \wedge j > i \wedge i > q$

Table 1: Domains of the places of the CPN in Fig.1 .

tions in the TPN as the cardinality of the Cartesian Product of the domains of its input places. Therefore, each transition in the TPN is associated with a unique combination of input places of the Cartesian Product. Only guards with true values produce results. By construction, each unfolded transition represents an individual event that is associated with a single combination of tokens.

The unfolding of a CPN into a TPN generates a simpler but semantically equivalent net and is the key to solve the gap problem between model and execution cited before. Since Petri Nets are good to model a parallel process, the unfolded net allows to analyze restrictions and other conditions of the parallel execution of the algorithm.

Other key factor to model with Petri Nets is the execution of the parallel algorithm using the unfolded net. Each transition in the TPN represents univocally a task, and each input place represents univocally a data block used as parameter of this task, thus, the traditional concept of firing a transition in the PN model is related with the execution of a routine in a computer.

The unfolded net of Fig.1 is shown in Fig. 2, with a tile division of four ($n = 4$). The similarity with a Dependency Graph can be observed, but this latter only represents dependency between tasks without information of the elements that determine its dependency, in our case, the blocks of data. The Dependency Graphs also lack of two elements present in a PN, as they are, the concept of execution and the generality of the model.

To execute the algorithm, the PN-PEM framework uses a net developed in this way. It also needs the configuration of the processors to be used in the execution and the relations of the transitions with the tasks and the tokens with data blocks. Thus, once the algorithm is modeled with the CPN, it can be executed in parallel by a series of processors taken as parameters by the framework [16]. The execution details are explained in the next section.

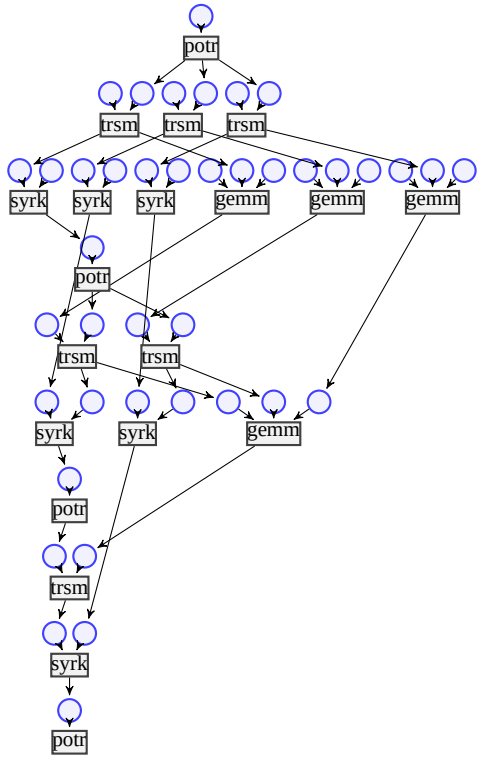

Figure 2: Token Petri Net unfolded from the Colored Petri Net in Fig.1 using a tile divisions $n = 4$.

3 The PN-PEM execution model

The TPN produced by the CPN unfolding as explained in the previous section, is used as input for the parallel execution of the framework, but is only one element of the set needed. Others elements are the set of logical processors used run the program in parallel, the relation between transitions and tasks to execute, and finally, the relation between tasks and kernels. The behavior is briefly explained in this section and all details can be seen in [17].

The framework uses the matricial form of the TPN [8], which is composed by P, T, I^-, I^+, M, where:

- P is a set of places P_i, with cardinality $|P| = p$.
- T is a set of transitions T_j, with $|T| = t$.
- 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, $p \times 1$.

Then, the PN-PEM model is conformed by:

- The unfolded TPN represented by its matricial form.
- A set of logical processors Π.
- One Mutual Exclusion mechanism χ .
- A set of tasks S.
- A set of executable kernels K.
- A function τ between transitions and tasks $\tau : T \rightarrow S$.
- For each logical processor Π_i there is a function γ_i between the tasks and kernels, $\gamma_i : S \rightarrow K$.

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. The mutual exclusion mechanism, χ, is used to guarantee the exclusion when updating M.

The set S is the set of Tasks that the algorithm must perform. τ is a function that relates the set of transitions T with the set of Task S. For each Processor Π_i there is a function γ_i that relates a task $s \in S$ with a *kernel* in K executed by the Processor.

The PN-PEM is very close to Timed Petri Nets [14]. 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. In PN-PEM, the firing action represents the execution of a task, but the semantics of the firing is different than in Timed Petri Nets, because firing is done by an idle Processor Π_i that selects a transition to fire among the enabled ones. After the transition is selected, the Processor takes its related task from S and launch the kernel defined by the function γ_i. In other words, the Processor is the responsible of the execution.

The number of enabled transitions can be lower or higher than the number of Processors. As a result of this, there may be 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 speed up will be poor, and this situation must be avoided. In the second case, the Processor must select the appropriate transition to fire. To do this, a function that prioritizes transitions must be implemented by the programmer and assigned to each processor. This function defines the priority of the available transitions to select the next kernel to be launched. It may not be unique for all processors, defining it properly to each processor.

The framework needs one Mutual Exclusion (mutex) mechanism to avoid concurrent reading and writing operations over vector M, because it is updated after each transition firing. In this sense, the Processors act serially when selecting the next transition to fire, waiting for the mutex enabled. The vector M represents the existence of tokens in each place, defining in our case, the algorithm state.

The Pseudo-code of the PN-PEM execution algorithm is depicted in Fig. 3. In round-robin style, each idle Processor Π_i makes the search for the task to perform by computing available transitions through matricial operations on I^- and M (step 3). The next step weighs up the available ones, by applying the priority function to the set of enabled

transitions choosing the transition with highest value, T_k (step 4).

```
 1. While main algorithm not finished
 2.   If can hold the mutual exclusion
 3.      Get the available transitions
 4.      Evaluate to select one to execute (Tk)
 5.      Update M absorbing Tk's input tokens
 6.      Free the exclusion
 7.      Task execution
 8.      Inject tokens in M
 9.   Else
10.      Delay
11.   Endif
12. End
```

Figure 3: Pseudo-code of the task selection algorithm.

The priority function takes as input parameter the Mark Vector and the Incidence matrices. In consequence, it evaluates the best selection according the state of computations. Data locality is also considered by using an stack that preserves data of the last task done. Additionally, as each logical processor can have a different valuation function, the scheduling policy is dynamic in the sense that the priority depends on the state of computations and the kind of processor involved. This design allows a flexible configuration of the framework in heterogeneous computers and is the key for achieve high performance in these systems.

Continuing with the execution algorithm, steps 5 and 8 of the pseudo-code represent 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 into their output places and potentially new transitions become enabled.

4 Experiments

The PN-PEM framework was developed in Fortran for shared memory computers using OpenMP directives to run parallel threads. It reads from the configuration files the TPN to execute, the Processors configuration, the mapping between tasks and transition, and the kernels to execute.

The computer used in our experiments is a dual socket Intel Xeon server, with two Xeon E5-2630 six core processors, 2.3 Ghz clock, performing twelve AVX 256 bits floating point units, and 64 GB RAM. Also, they are available two Xeon Phi 31S1P coprocessors, 57 cores at 1.1 Ghz, 8 GB RAM per card. The theoretical Rpeak for each type of processor are 441 Gflops for the CPU processors and 2001 Gflops for each coprocessor, single precision, and the half for double precision.

The Intel's suite Composer XE 2013 was used as the Fortran compiler and the MKL library of this suite, as the BLAS implementation. For each of the routines needed to execute in the Cholesky algorithm, i.e. xPOTRF, xGEMM, xTRSM and xSYRK, two kernels where coded, one for the CPU's processors and other for the Xeon Phi processors. For the laters, the offload mode was used.

Tests done involve four different ways of processors utilization, differing in the way the CPU cores are used alone or combined with the coprocessors, and in the form to grouping the cores. For the CPU cores, the -xAVX compiler option was always used, because it uses the AVX unit provided by the processor, which is specific to improve the performance of the floating point operations.

Thereby, in the used computer, a maximum of twelve parallel tasks can be executed on the CPU. They were grouped in one logical processor composed by twelve cores or in two groups of six cores, dividing them by affinity of hardware. These are called in the rest of the paper as "CPU based processors".

In both cases, the parallelism at internal level of the CPU based processor is managed by the parallel implementation of the MKL library of the routine called by the kernel. At a high level, the parallelism is managed by the PN-PEM framework, according the scheduling algorithm explained before. On the other hand, for the coprocessors, the whole board is used as a logical processor at the PN-PEM level, and the internal parallelism is derived to the MKL implementation for the Xeon Phi devices.

In this way, the four configurations defined for tests were 1×0, 2×0, 0×2 and 2×2, called according the number of logical processors of each type used, the first value for the number of "CPU based processors", and the second for the number of coprocessors used. Thus, the 2×2 is the configuration that use jointly both kind of physical processors, grouped by two "CPU based processors" of six cores each, and two Xeon Phi cards.

Tests were done with a size of matrix that justify the use of the joint computing resources, i.e., ranges of 48000, 60000, 74000 and 96000, for single and double precision. The number of tile divisions was fixed at eight, namely, the matrix is divided in 64 square blocks, eight rows by eight columns. This number of tile division defines 120 tasks, with a series of 19 sequential tasks [16] and determine a data division with a block size that has a good processor performance, while the number of tasks is enough to get a large number of parallel tasks.

A number lower than eight in the tiled data division improves the performance of an individual processor but provides fewer tasks to run in parallel, and, in opposition, more than eight divisions, increase the parallel possibilities but decreases the performance of processors, with an overall outcome with fewer gflops.

The most remarkable results of tests are shown in Tables 2 and 3, for single and double precision numbers respectively. The best results are obtained for the larger matrix range, as expected.

The first two columns of both Tables, the configura-

Prcs.	1×0		2×0		0×2		2×2	
Size	secs	flps	secs	flps	secs	flps	secs	flps
48K	98.8	385	91.8	417	66.8	570	48.3	788
60K	200.1	369	172.5	428	108.6	680	80.7	914
72K	308.0	412	295.9	429	165.0	770	134.1	947
84K	502.4	400	460.5	437	264.8	760	193.1	1042
96K	850.8	352	671.0	446	360.0	832	268.3	1116

Table 2: Time in seconds and Gflops observed for single precision tests.

Prcs.	1×0		2×0		0×2		2×2	
Size	secs	flps	secs	flps	secs	flps	secs	flps
48K	201.3	189	180.4	211	144.3	264	100.8	378
60K	409.5	180	346.9	213	265.2	278	184.7	400
72K	755.6	168	594.9	214	381.8	333	287.6	442
84K	997.8	202	945.1	213	562.9	357	435.3	462

Table 3: Time in seconds and Gflops observed for double precision tests.

Block Size Pres Proc	10.5K SGL CPU	10.5K SGL Phi	9K DBL CPU	9K DBL Phi
trsm	7.37	4.31	12.30	7.21
syrk	7.92	2.86	10.46	4.37
gemm	12.82	5.43	17.69	7.86

Table 4: Observed time for the kernels executed over the CPUs (six cores) and the coprocessor, in seconds.

tions 1×0 and 2×0, correspond to the tests that use twelve cores as one "CPU based processor" or as two; in the later case, assigning six cores to each of these.

Analyzing the results of these columns, it can be seen that the logical division of cores provides more Gflops, thus, using the PN-PEM framework with the MKL library brings better performance than using the library alone. An explication of this fact is the bigger number of cache fails when using all the cores to perform a single task than when dividing the cores as two processors, respecting the physical affinity.

The internal logs of tests point out that the execution of a any task, takes more time in a "CPU based processor" than in the coprocessor, as expected. The Table 4 shows the average time for the tasks executed on the CPUs (six cores) and on the coprocessor for two cases of different block range and precision. In both cases, for the GEMM routine, which is dominant in the computations, takes more that the double of time to execute on CPU based processors than in one Xeon Phi coprocessor. This is the effect of the heterogeneity of processors.

The PN-PEM model allows two ways to solve the heterogeneity problem: by a dynamic scheduler that launches tasks when processors become idle, or by a double granularity, assigning smaller data blocks to slower processors.

In the case of this work, the solution is to assign dynamically tasks to the "CPU based processors" with the same granularity than for the coprocessors, but respecting its rythm and the advance of the computations.

Consider the relative velocity of faster processors with respect to slower. In our case, it is considerated that coprocessors can be two times faster than CPUs, rounding numbers. When the number of available tasks is greater than two, i.e., there are three or more tasks available, the slower processor can execute a task without interfere in the overall performance selecting a task out of the "critical path" of execution. In the other case, when there are one or two available tasks, the slower processor must remain idle and leave the execution of these task to the coprocessors. This feature is implemented in the PN-PEM framework by configuring the priority function for each type of processor, as explained before.

Coming back to Tables 2 and 3, it can be seen that the computing power of each coprocessor is almost the double of the "CPU based processors", for single and double precision. The utilization of both type of processors, CPU cores and Xeon Phi jointly, does not scale linearly. Nevertheless, the Gflops obtained are almost the sum of both: the sums of the Gflops of the tests 2×0 plus the 0×2 for different matrix sizes and numerical presicion, it is between the 80% and 90% of the value of the corresponding test for 2×2.

The last result was obtained after a series of tuning tests by changing the valuation function of the processors, prioritizing the next task to execute with different criteria.

The Fig. 4 shows a timeline of an execution. It shows the results of the test for 2×2 processors with a matrix range of 84000, single precision. The upper two lines represent the execution of coprocessors while the lower two the execution of CPUs. White spaces represent the processors idleness. Can be noted that they exist only at the beginning and at the end of the execution, and they are nonexistent when there are tasks available to run in parallel.

In tests where both types of processors are used jointly, each type can execute any of the four tasks, implementing them by proper kernels to each architecture. In last stages of the execution, the processing becomes almost sequential, as is show in Fig. 2, fact also noted in the final phase of execution in Fig. 4. There is seen that only coprocessors execute the last tasks, leaving inactive the "CPU based processors".

Also can be remarkable that the real performance reached by the "CPU based processors" is near its Rpeak, and for the coprocessors, it is just a quarter. For the former, the hyperthreading technology allows to use the processor in a task while another task is awaiting for data coming from the main memory. For the later, the Rpeak does not consider the time involved in the copy of data from main memory to the memory of the board, which involves a pass by the pci channel, which takes most of the elapsed time.

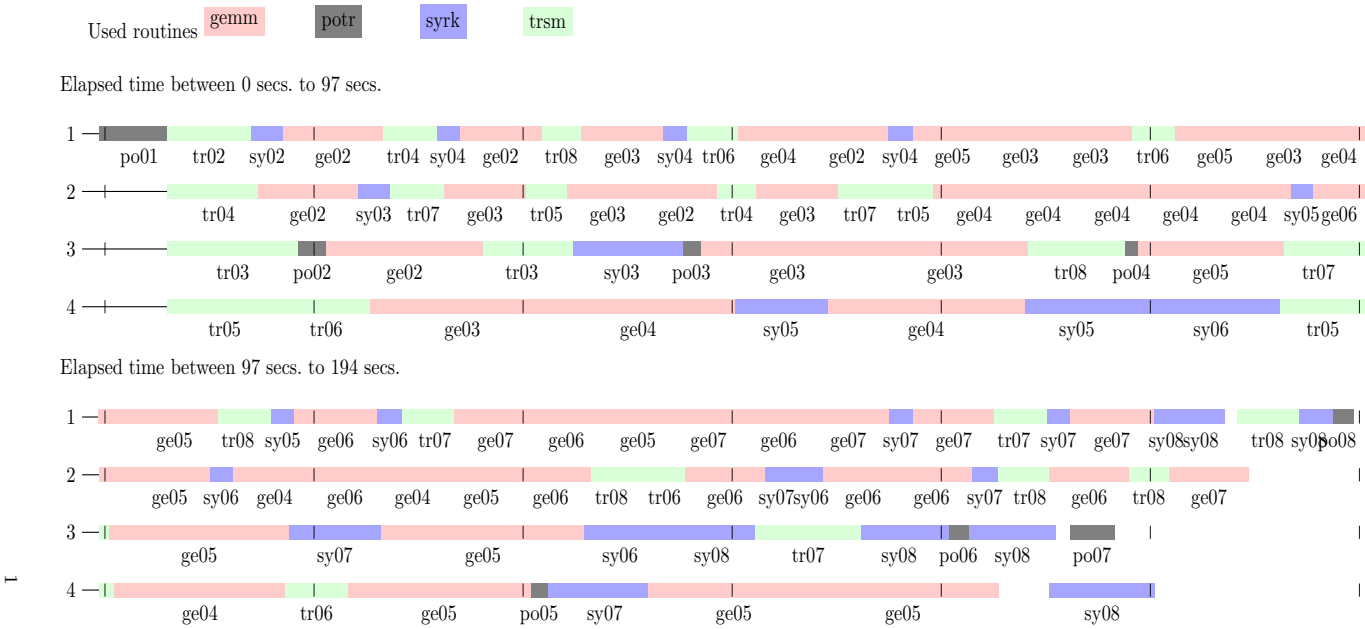

Figure 4: Execution timeline, divided in two sections, 2×2 processors, 84000 range, single precision. Timelines 1 and 2 represent the coprocessors, and timelines 3 and 4, "CPU based processors" (six cores each).

5 Related Works

The PN-PEM scheduler is related to the dynamic scheduler of the Quark project [18]. This last prioritizes data locality as a policy of improvement performance. Instead, the PN-PEM design allows to define a policy not only restricted to data locality, but instead, to the availability of tasks to run in parallel. Also, each Processor may adapt a scheduling policy appropriate to its performance, allowing more than one scheduler for system.

StarPU is a runtime system developed at the INRIA Institute that launches tasks in parallel over a set of processors, using a dynamic scheduler [1]. It is based on kernels provided by the user that implement the solution appropriate to each processor. The scheduler uses estimated time to select the task to execute. The definitions of tasks, dependencies and data partition must be coded by the programer. Changes in any of these involve changes in the code, which is a 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. The scheduler is dynamic following a FIFO order without considering any other factor of optimization.

Shetti et.al. implements the HEFT (Heterogeneous Earliest-Finish-Time) scheduling algorithm in a CPU-GPU environment [11], which is similar to the scheduler implemented in the PN-PEM framework. It has the drawback that the assignment of tasks priorities is done before running.

A recent work introduces the implementation of the MAGMA library [12] using Xeon Phi cards [5]. It uses a DAG to determine task dependencies, nevertheless, it is not explained how is the criteria and the algorithm to select the next task when there are many available. By other side, the scheduling policy directs tasks in the "critical path" to CPUs, and the remaining to the accelerator, differing with the showed in this paper.

6 Conclusions

As a continuation of the previous work, tests were extended to an heterogeneous parallel machine without a big effort. It was needed to configure the type of processor used in the computations and its corresponding kernels. In order to optimize performance, two different priority functions were assigned, one for the "CPU based processors" and other for the Xeon Phi cards.

Test gave results that ensures that the main objectives of this study were achieved. The performance reached is very close to the aggregate power of computation of the two types of processors used, as is established by the objectives of the work. This fact was a result of tuning the dynamic scheduler of the PN-PEM framework according the algorithm, the state of the computations and the properties of the machine used.

Compared with related systems, at the best of our knowledge, the PN-PEM model and framework is more flexible and quickly adaptable to changes. The model helps to analyze, execute, and tune the execution, with a negligible overload.

Future work will focus on applying the PN-PEM model and framework with others algorithms. Also will be

considered to extend the framework to a distributed memory architecture.

References:

[1] Cedric Augonnet, Samuel Thibault, and Raymond Namyst. Starpu: a runtime system for scheduling tasks over accelerator-based multicore machines. Technical Report 7240, INRIA, March 2010.

[2] SJ Bachega and DM Tavares. Applications of petri nets in distributed processing: a scoping study. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 109. The Steering Committee of WorldComp, 2015.

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

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

[5] Jack Dongarra, Mark Gates, Azzam Haidar, Yulu Jia, Khairul Kabir, Piotr Luszczek, and Stanimire Tomov. Hpc programming on intel many-integrated-core hardware with magma port to xeon phi. *Scientific Programming*, 2015, 2015.

[6] Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, and Bruno Raffin. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Boston, Massachusetts, États-Unis, May 2013.

[7] Azzam Haidar, Hatem Ltaief, Asim YarKhan, and Jack Dongarra. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. Technical Report 243, LAPACK Working Note, March 2011.

[8] Marian Iordache and Panos Antsaklis. *Supervisory Control of Concurrent Systems: A Petri Net Structural Approach*. Birkhäuser Boston, Boston, 2006.

[9] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[10] Hatem Ltaief, Stanimire Tomov, Rajib Nath, Peng Du, , and Jack Dongarra. A scalable high performant cholesky factorization for multicore with gpu accelerators. Technical Report 223, LAPACK Working Note, November 2009.

[11] Karan R. Shetti, Suhaib A. Fahmy, and Timo Bretschneider. Optimization of the heft algorithm for a cpu-gpu environment. In *Proceedings of the 2013 International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '13, pages 212–218, Washington, DC, USA, 2013. IEEE Computer Society.

[12] the University of Tennessee. The MAGMA project. Matrix Algebra for GPU and Multicore Architectures. http://icl.cs.utk.edu/magma.

[13] Top500.org. Top 500 supercomputers sites. http://www.top500.org/.

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

[15] Gustavo Wolfmann and Armando De Giusti. Parallel asynchronous modelization and execution of cholesky algorithm using petri nets. In *The 2013 Internat. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA13)*, 2013.

[16] Gustavo Wolfmann and Armando De Giusti. Petri net based algorithm modelization and parallel execution on symmetric multiprocessors. In *The 2014 Internat. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA-14)*, 2014.

[17] Gustavo Wolfmann and Armando De Giusti. The pnpem framework: a petri net based parallel execution model. *Journal of Computer Science & Technology*, 15(2):129–136, November 2015.

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