Kronecker Descriptor Partitioning for Parallel Algorithms*

Ricardo M. Czekster, Cesar A. F. De Rose, Paulo Fernandes
Antonio M. de Lima, Thais Webber
PUCRS – Av. Ipiranga, 6681– Porto Alegre – 90619-900 – Brazil
{ricardo.czekster, cesar.derose, paulo.fernandes, antonio.lima, thais.webber}@pucrs.br

Keywords
Kronecker Products, Data Partitioning, Parallel Algorithms.

Abstract
The key operation to obtain stationary and transient solutions of transition systems described by Kronecker structured formalisms is the Vector-Descriptor product. This operation is usually performed with shuffling operations and matrices aggregations to reduce the floating point multiplications inside iterative methods. Due to the flexibility of the *Split* method treating Kronecker product terms, it is a natural alternative to decompose descriptors within parallel environments. The main problem is to define the correct task size to assign to each node and also the shared memory size, since sending a small task per time can lead to a larger communication overhead. In this paper we are investigating data partitioning strategies for a parallel solution of transition systems obtained from Kronecker descriptors using the Split algorithm.

1. KRONECKER-BASED DESCRIPTOR
Kronecker-based descriptors are used to represent complex transition systems in a memory-efficient way. A very common application of such structure is to describe huge Markovian systems, mitigating the state space explosion problem often associated to these representations. These non-trivial structures uses tensor algebra to manipulate its inner low dimension matrices to be used in performance and reliability evaluation. A myriad of structured formalisms [1] is available to the research community, *e.g.*, Stochastic Petri Nets (SPN), Performance Evaluation Process Algebra (PEPA), and Stochastic Automata Networks (SAN), to name a few. The Kronecker descriptor, or simply descriptor, maps an underlined Continuous Time Markov Chain [7] representing an infinitesimal generator. The main idea of a Vector-Descriptor Product (VDP) procedure is to multiply a probability vector by a non-trivial structure, the descriptor, in order to achieve (if possible) a stationary regime and subsequent performance indices. For the set of formalisms equipped with tensor manipulations, the numerical processes deal with vector products employing the classic Shuffle algorithm [5, 6] to implicitly access the underlined transition matrix.

A descriptor is decomposed in tensor product terms [6], where each term corresponds to a set of small matrices and tensor product operators. VDP can be summarized in multiplying a probability vector π to a descriptor, *i.e.*, $\sum_{j=1}^{N+2E}\left(\pi \times \left[\bigotimes_{i=1}^{N} Q_j^{(i)}\right]\right)$. To efficiently perform VDP, specialized algorithms are proposed throughout the years, namely the traditional Shuffle Algorithm and the flexible Split Algorithm [3]. The main differentiation between the algorithms concerns the additional memory requirements and the computational cost in terms of needed multiplications.

Figure 1 graphically presents the Split idea [3] when applied to the multiplication of a vector by a generic tensor term. The algorithm affects all Kronecker product terms inside a descriptor, at each iteration of a numerical method [7] (*e.g.* Power method, Arnoldi, GMRES). The algorithm considers a given cut-parameter σ separating the tensor term in two different sets of matrices. In the example, $\sigma = N - 2$ is indicating the end of a sparse-like part (on the matrix $Q_j^{(N-2)}$), which means this part is selected to perform nonzero elements combinations, *i.e.*, aggregating these matrices. The rest of matrices are selected to be treated with shuffling operations maintaining the tensor structure.

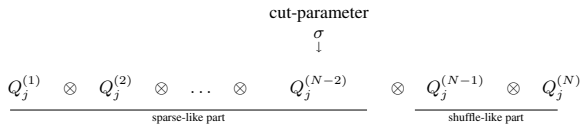


Figure 1: Split method with a flexible cut-parameter σ

Algorithmically, Split [3] consists of the computation of scalars or AUNFs (Additive Unitary Normal Factors) by multiplying one nonzero element of each matrix by the matrices at left of σ (from $Q_j^{(1)}$ to $Q_j^{(\sigma)}$). According to the elements row indexes used to generate the AUNFs, a contiguous slice of the input vector is taken to be multiplied by this scalar. The resulting vector is used as input vector to the Shuffle-like multiplication by the tensor product term at right of σ (from $Q_j^{(\sigma+1)}$ to $Q_j^{(N)}$).

Figure 2 shows the operations flow in a sequential implementation of the Split method. The first steps (a) and (b) are executed

*Authors are supported by Petrobras (0050.0048664.09.9). Paulo Fernandes is also supported by the Brazilian government (CNPq 307272/2007-9). The authors thank Felipe M. Franciosi from Imperial College London (UK) and Pedro M. Velho from UJF (France) for their development efforts in earlier parallel approaches. The order of authors is merely alphabetical.

to initially setup the method putting both matrices and the probability vector in memory. The step (c) referred to the creation of scalars using the matrices at left of the σ to aggregate nonzero values. These nonzero values are stored in memory (not necessarily because they can be generated at runtime, used and then discarded, but this on-the-fly generation increments the numerical operations involved in the multiplication) with information about their related indexes in the vector. Steps (d) and (e) are correspondent to the processing core where the massive iterative dynamic for huge models actually occurs, *i.e.* with massive state spaces (until approximately 100 million states).

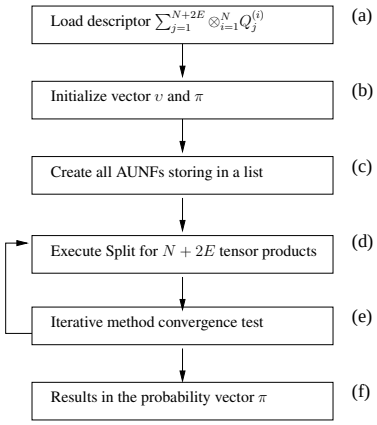


Figure 2: Sequential solution for the Split method

Recent developments [4] indicate that reordering the matrices on the tensor product terms, one can produce faster results for some cases due to numerical complexity reduction, although the additional permutation costs. Moreover, due to the flexibility of the *Split* method treating Kronecker product terms, it is a natural alternative to decompose descriptors within parallel environments. The main problem is to define the correct task size to assign to each node and also the shared memory size, since sending a small task per time can lead to a larger communication overhead. On the contrary, transferring a large task can generate unfair load balance. For the specific case of Split, a potentially huge vector have to be manipulated while tasks must be assigned to the set of available processors and possible large communications are derived.

2. DESCRIPTOR PARTITIONING

Numerical solutions are often analyzed to carry out calculations simultaneously, considering that large problems could be divided into smaller ones, with or without data dependency. High performance computing using parallel processing depends on efficient problem partitioning and load balance. Partitioning comprises balancing the computational load and overheads created by data communication on processors in order to minimize the total parallel computation time. In this paper we investigate data partitioning strategies for a parallel solution of transition systems obtained from Kronecker descriptors, implementing the Split algorithm. We are using clusters as the target architecture running a *master-worker* [8] execution scheme. The idea is to distribute a set of matrices to each node and execute the multiplications concurrently. Each node takes some time processing its own task, and returns the resulting vector as soon as possible to the master process.

Figure 3 shows the operations executed by the master and the worker processes. All processes execute all steps but one, related to the iterative process (f.1 and f.2). In general, each process will be responsible for an interval of tasks which means Kronecker products to multiply (d) and generated AUNFs (c), in different ways. Also the intermediary results are consolidated in the probability vector (e) in which the master process has access to verify convergence and control the workers (f.1). At the same point, workers wait for the master to decide if they should continue the multiplication process or terminate (f.2).

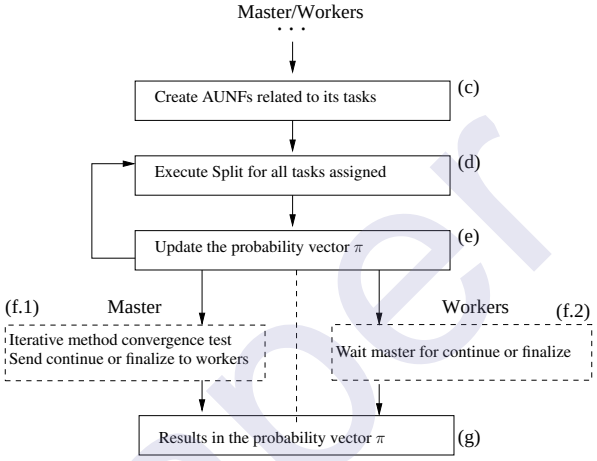


Figure 3: Parallel solution method

Note that all workers have one local copy of the descriptor as seen in Figure 2 - step (a), and given the σ established for each tensor product, some slices of vectors are initialized in each worker as step (b). Due to this steps (a) and (b) are omitted in this last figure. Because a Kronecker descriptor is composed of tensor product terms, these are natural candidates for the partitioning. However, Kronecker product terms can be exploited in different ways. The Split algorithm [3] allows each tensor product to be partitioned into smaller work units called AUNFs that can be distributed among processors. Note that the step (c) suffers a great impact in different partitioning approaches, *i.e.*, by Kronecker product terms itself and by AUNFs quantities, mainly because tensor product terms can strongly vary matrices types, dimensions, sparsity, and all these characteristics determine the cut-parameter and consequently the total number of AUNFs.

2.1 Partitioning per Kronecker Product

One partitioning approach is based on the total number of Kronecker products and available processors, *i.e.*, a set of tensor product terms are the bag of tasks to be distributed. The computational cost in multiplications related to each tensor product term in each node is given by $\left(\prod_{i=1}^{\sigma_j} nz_j^{(i)}\right)\left(\prod_{i=\sigma_j+1}^{N} n_j^{(i)}\right)$, where $nz_j^{(i)}$ corresponds to the total number of AUNFs in a term j and $\prod_{i=\sigma_j+1}^{N} n_j^{(i)}$ is the size of the vector to be multiplied. Additionally, the theoretical communication cost of a Kronecker product in each processor follows the sum of the size of the slice vectors to be multiplied by each AUNF in the iteration is given by $\sum_{k=1}^{nz_j^{(i)}}\left(\prod_{i=\sigma_j+1}^{N} n_j^{(i)}\right)$. The size of these messages will vary according to the number of processes allocated and the cost per AUNF for execution. One can have as many processors or nodes as the total number of Kronecker products, or even divide quite equally the work to be done among np nodes.

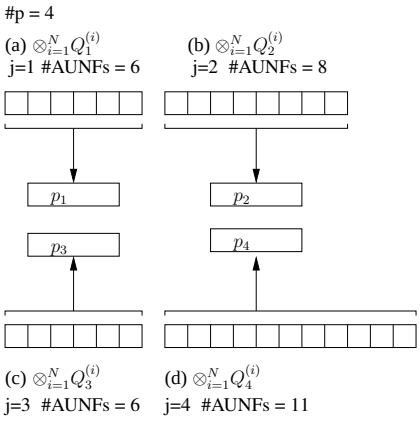


Figure 4: Partitioning per Kronecker product

Figure 4 illustrates a model with four Kronecker products (#tprod=4, j=1...4), each one generating it own list of AUNFs, assigned among processors, without take into account the load balancing related to the number of AUNFs actually generated by each one. Steps (a),(b),(c) and (d) are iterative assignments of Kronecker products to each processor.

It is possible if the number of Kronecker products is greater than the number of processors, some processors receive more load to compute than others. However, this approach derives bottlenecks related to the load balancing in the nodes because each Kronecker product can have very different number of AUNFs to be processed.

2.2 Partitioning per AUNFs

A different partitioning approach is to distribute the computation of each AUNF, or a set of them, to each node. Figure 5 illustrates the second partitioning approach which is based on a load balance that tries to assign equally sized tasks to each node, as fair as it can. The bag of tasks is actually the total number of generated scalars (AUNFs) considering the entire descriptor, and those are sufficiently independent to be multiplied by different processes, in different vector positions and dimensions. The dimensions are related to the indexes calculated to each scalar when preprocessing the matrices at the left-hand side of σ. In this example, the

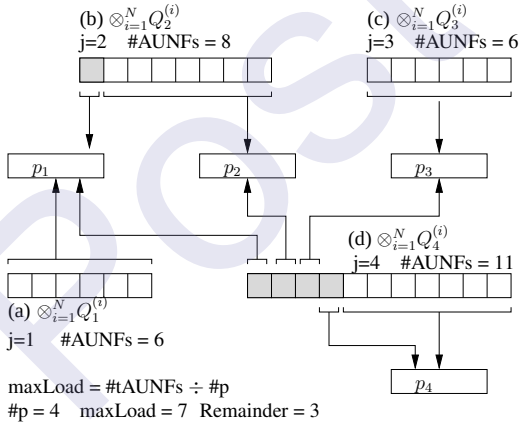


Figure 5: Partitioning per AUNFs

model presents four Kronecker products (j=1...4), each one generating separated lists of AUNFs. The total number of AUNFs for each product is given by the variable #AUNFs as seen in Figure 5.

The parallel execution illustration demonstrates the partitioning approach by AUNFs in four processors (#p=4). The algorithm proceeds first calculating a maximum load (maxLoad) per processor, simply dividing the total number of AUNFs (tAUNFs) by the total number of processors (#p). The remaining of this division is also divided by the processors after the first assignment of AUNFs. The steps are indicated as follows: (a) the first Kronecker product (j=1) presents six AUNFs then it fits well in the first processor (p_1) because the maximum load calculated is superior; (b) the second Kronecker product (j=2) has eight AUNFs then seven AUNFs are assigned at p_2 remaining one for later assignment; (c) the next Kronecker product (j=3) presents also six AUNFs been directly assigned to processor p_3; (d) finally, the last Kronecker product (j=4) generates eleven AUNFs placing seven of them at processor p_4, remaining four AUNFs.

After these steps the algorithm sums the last AUNFs considering all Kronecker products (in the example it has still five AUNFs), and divide the AUNFs among processors as equally as possible, observing first the processors in which the maximum load is not already achieved (b) and after distributing the lasting tasks (d). On certain occasions, a given processor can have more tasks than others however, in general, the load balancing works favorably well. The parallel computation cost will be determined by the large amount of AUNFs and vectors to update in a processor.

3. PRELIMINARY RESULTS

The Split algorithm in both sequential and parallel versions could be applied to any other modeling formalism that present a Kronecker descriptor as structured representation. This numerical analysis is carried out to identify bottlenecks and drawbacks of parallel implementations of iterative methods performing vector-descriptor product. The tests were executed in a cluster architecture containing seven homogeneous nodes connected in a Gigabit Ethernet network. Each node is composed of two processors Intel Itanium^2 (Madison) 1.5GHz, 2GB memory, L3 cache memory 6MB, under Linux O.S. The prototype was compiled using *gcc* version 4.2.4 and MPICH library version 1.2.7p1.

Table 1: Kronecker product terms in the descriptor

j	Total of AUNFs	Extra mem. (Kb)	Total of Mults.	Time (s)
[1:21]	21	0.33	32,280,161	3.81
[22:22]	2,421,009	37,828.27	2,421,009	1.86
[23:23]	40,960	640.00	40,960	0.03
[24:33]	400	6.25	7,873,200	0.64
[34:43]	800	12.50	47,239,200	3.47
Total time spent in one iteration (s)				10.48
Total iterations	2696	**Total time spent (min.)**		470.90

Table 1 shows that for a SAN model [2] containing *e.g.*, ten automata representing workers (N=10) and a buffer automaton of forty positions (K=40), is generated a total of 43 Kronecker product terms in the descriptor to be multiplied by a vector. For each Kronecker product term j, a different σ is determined resulting in various sets of AUNFs. Note that the sequential execution time of this model using Split is approximately 6.07 hours.

The partitioning of Kronecker products was balanced enough (Table 2), however, the number of AUNFs generated in each node was very unfair considering the total number of multiplications per node. Table 3 shows the range of AUNFs per Kronecker product fairly partitioned among two processors. Note that the total number of AUNFs in each one is the same consequently the memory spent. However the different sets of AUNFs present varied sizes of vectors

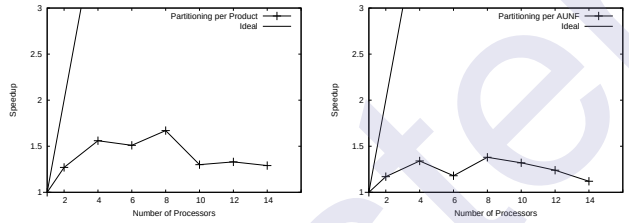
Table 2: Partitioning per Kronecker product

Process $p = 1$				
Kronecker products	Total of AUNFs	Extra Mem. (Kb)	Total of Mults.	Time (s)
22	2,421,620	37,837.81	46,117,311	5.83
Process $p = 2$				
Kronecker products	Total of AUNFs	Extra Mem. (Kb)	Total of Mults.	Time (s)
21	41,570	649.53	43,737,220	3.99
Size of vector π				7,263,027
Time of the update phase (s)				1.60
Total time spent in one iteration (s)				8.10
Total iterations	2696	**Total time spent (min.)**		363.96

Table 3: Partitioning per AUNFs

Process $p = 1$					
Kronecker products	Range of AUNFs per term	Total of AUNFs	Extra Mem. (Kb)	Total of Mults.	Time (s)
10	[1:1]	10	0.16	16,140,060	1.90
1	[1:1231585]	1,231,585	19,243.52	1,231,585	0.99
Total		1,231,595	19,243.67	17,371,645	3.89
Process $p = 2$					
Kronecker products	Range of AUNFs per term	Total of AUNFs	Extra Mem. (Kb)	Total of Mults.	Time (s)
11	[1:1]	11	0.17	16,140,101	1.85
10	[1:40]	400	1.56	7,873,200	0.63
10	[1:80]	800	12.54	47,239,200	3.46
1	[1:40960]	40,960	640	40,960	0.03
1	[1231586:2421009]	1,189,424	18,584.75	1,189,424	0.95
Total		1,231,595	19,243.67	72,482,886	6.92
Size of vector π					7,263,027
Time of the update phase (s)					1.62
Total time spent in one iteration (s)					8.94
Total iterations					2696
Total time spent (min.)					401.70

to multiply. This characteristic provokes different computational costs in the nodes despite the same load assigned. Notably, for this class of model the partitioning per Kronecker product term seems to be better without a correct analysis of the second approach. This means to consider more variables such as vectors size versus total of AUNFs to obtain gains in the parallelization.



(a) Section 2.1 approach (b) Section 2.2 approach

Figure 6: (a) Partitioning per Product, (b) Partitioning per AUNFs

The Figure 6 shows the parallel results for the model presented in both partitioning strategies. The best gains in terms of performance were verified when the partitioning approach by Kronecker products is used as shown on Tables 2 and 3, improving the workload distribution between the cluster nodes. The speed up curve began to degrade from 8 processors due to communication costs and vector updates that increased proportionally to the number of processes. Moreover, the obtained gain executing more tasks in parallel was not covered by the high transmission cost of vectors.

4. CONCLUSION

Theoretical performance analysis of parallel implementations shows that some approaches can significantly decrease the execution time of most sequential algorithms. The major attraction to use the Split algorithm concern its flexibility, offering more effective parallelization alternatives since the multiplication can be executed independently of the result of other Kronecker product terms, or even other AUNFs. In a parallel algorithmic version, a node could receive a bag of tasks with pre-computed scalars to multiply by the vector π.

As mentioned in the paper, there are many issues to address in future works concerning communication, memory costs and workload distribution. For example, it seems natural to verify not only the total number of AUNFs to compute, but also take in account the number of required multiplications in each Kronecker product, in order to produce a more efficient load balancing. Or, for that matter, it seems right to admit that even the adequate cut-parameter in a sequential implementation may not be necessarily a good choice for parallel implementations. The current study is clearly concerned with communication issues mainly because the size of vectors to update can vary depending on the chosen σ value. Shifting σ to the right position in each Kronecker product term causes the reduction of the vectors size to be updated. This diminishes the communication costs and not necessarily increases the quantities of AUNFs necessary to perform the task.

A more demanding future work consists taking in account the fact that memory constraints can be overcome using other paradigms with shared memory for the vectors and different load balance approaches. A hybrid approach using MPI and OpenMP could enhance the speedup of a parallel solution reducing the communication and memory costs in clusters with multicore architectures. Obviously, the work presented in this paper will benefit from a further investigation of communication costs and better balanced choices to distribute the available tasks. At least the working prototypes of Split parallel implementation already allow us to foresee a performance increase when solving large transition systems stored and manipulated as descriptors.

5. REFERENCES

[1] Formal Methods for Performance Evaluation, SFM 2007, Bertinoro, Italy, Advanced Lectures. In M. Bernardo and J. Hillston, editors, *SFM*, volume 4486 of *LNCS*. Springer, May/June 2007.

[2] L. Baldo, L. Brenner, L. G. Fernandes, P. Fernandes, and A. Sales. Performance Models for Master/Slave Parallel Programs. *Electronic Notes In Theoretical Computer Science (ENTCS)*, 128(4):101–121, April 2005.

[3] R. M. Czekster, P. Fernandes, J.-M. Vincent, and T. Webber. Split: a flexible and efficient algorithm to vector-descriptor product. In *ValueTools'07*, volume 321 of *ACM International Conference Proceedings Series*, 2007.

[4] R. M. Czekster, P. Fernandes, and T. Webber. GTAexpress: a Software Package to Handle Kronecker Descriptors. In *QEST2009*. IEEE Computer Society, September 2009.

[5] M. Davio. Kronecker Products and Shuffle Algebra. *IEEE Transactions on Computers*, 30(2):116–125, February 1981.

[6] P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor-vector multiplication in Stochastic Automata Networks. *Journal of the ACM*, 45(3):381–414, May 1998.

[7] W. J. Stewart. *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling*. Princeton University Press, NJ, USA, 2009.

[8] B. Wilkinson and M. Allen. *Parallel Programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, Upper Saddle River, NJ, 1999.