Speeding-up the Fault-Tolerance Analysis of Interconnection Networks

D. Bermúdez Garzón, C. Gómez, P. López and M. E. Gómez
Departamento de Informática de Sistemas y Computadores, DISCA
Universitat Politecnica de Valencia
Valencia, Spain
Email: dieberg1@posgrado.upv.es

***Abstract*—Analyzing the fault-tolerance of interconnection networks implies checking the connectivity of each source-destination pair. The size of the exploration space of such operation skyrockets with the network size and with the number of link faults. However, this problem is highly parallelizable since the exploration of each path between a source–destination pair is independent of the other paths. This paper presents an approach to analyze the fault-tolerance degree of multistage interconnection networks using GPUs in order to speed-up it. This approach uses CUDA as parallel programming tool on a GPU in order to take advantage of all available cores. Results show that the execution time of the fault-tolerance exploration can be significantly reduced.**

***Index Terms*—Fat-Tree, MINs, fault-tolerance, CUDA.**

I. Introduction

The analysis of fault-tolerance of highly-parallel machines is a topic of increasing interest. As the system grows, not only its processing capacity does, but also the probability of faults in the system. This probability linearly grows with the number of elements in the system. As the availability of these systems is a concern, it is critical to keep the system working, even in the presence of faults.

A key component of HPC systems from both the fault-tolerance and performance points of view is the interconnection network. It must provide efficient communication among the computing elements to achieve the highest system performance; but also, tolerating faults becomes a problem of great importance since faults in the interconnection network may isolate a large fraction of the machine, containing many healthy processors that otherwise could still be used. The main design parameter that impacts the fault-tolerance degree of an interconnection network is the topology.

The fault-tolerance concern is recurrent when designing a new topology as can be found in the literature [2]–[8], [10], [12], [14]–[18], but few of these works perform a deep fault-tolerance analysis to quantify the fault-tolerance level achieved by the proposed topologies [9]. These results are very useful when selecting the topology for a new machine. The main problem when performing those analysis is the required computational power. The exploration space size highly increases with network size and with the number of simulated faults.

To overcome this limitation, in this paper, we propose a methodology that takes advantage of GPU processing capabilities to implement highly parallel algorithms that allow the exploration of the whole space or, at least, a highly representative subspace of the topology fault-tolerance. In order to complete the exploration of the fault-tolerance of the topology, for each combination of faults, the connectivity of every pair of nodes in the system must be checked. The good news are that each source–destination pair for a given combination of faults can be checked independently of the rest, which provides a high potential for parallelization. Also, each combination of faults for a given number of faults can be checked independently. In particular, we apply this methodology to the fat-tree topology as an example, but it can be applied to any topology. The fat-tree is one of the most widely used topologies in large machines (see the Top500 list [1]) since it does not only provide a good performance/cost ratio but also a good fault-tolerance degree.

The rest of the paper is organized as follows. Section II provides some background on the fat–trees. It also describes the adaptive routing algorithm commonly–used in fat–trees. Section III provides some background on parallel computing. Section IV describes the problem of analyzing fault-tolerance of interconnection networks. Section V presents the proposed algorithm, and Section VI provides the results obtained with the new proposal and finally in Section VII some conclusions are drawn.

II. Fat–Tree Topology

The fat-tree topology is based on a complete tree that gets thicker near the root. The arity of the switches increases as we go nearer to the root, which makes the physical implementation unfeasible. For this reason, some alternative implementations have been proposed in order to use switches with fixed arity.

In particular, the k–ary n–tree [13] is a parametric family of regular multistage topologies. The number of stages is n and k is the arity or the number of links of a switch that connects to the previous or to the next stage (i.e., the switch degree is $2k$). A k–ary n–tree is able to connect $N = k^n$ processing nodes using nk^{n-1} switches. Each processing node is represented as a n–tuple $\{0, 1, ..., k-1\}^n$, and each switch is defined as a pair $\langle s, o \rangle$, where s is the stage where the switch is located at,

$s \in \{0..n-1\}$, and o is a $(n-1)$-tuple $\{0,1,...,k-1\}^{n-1}$ which identifies the switch inside the stage.

Two switches $\langle s, o_{n-2}, ..., o_1, o_0\rangle$ and $\langle s', o'_{n-2}, \ ... \ , o'_1, o'_0\rangle$ are connected by an edge if $s' = s+1$ and $o_i = o'_i$ for all $i \neq s$. On the other hand, there is an edge between the switch $\langle 0, o_{n-2}, ..., o_1, o_0\rangle$ and the processing node $p_{n-1}, ..., p_1, p_0$ if $o_i = p_{i+1}$ for all $i \in \{n-2, ..., 1, 0\}$. This edge is labeled with p_0. In what follows, we will assume that descending links are labeled from 0 to $k-1$, and ascending links from k to $2k-1$. Figure 1 shows a k–ary n–tree example. The upward links are shown in blue (dotted lines) and the downward links are shown in red color (dashed lines).

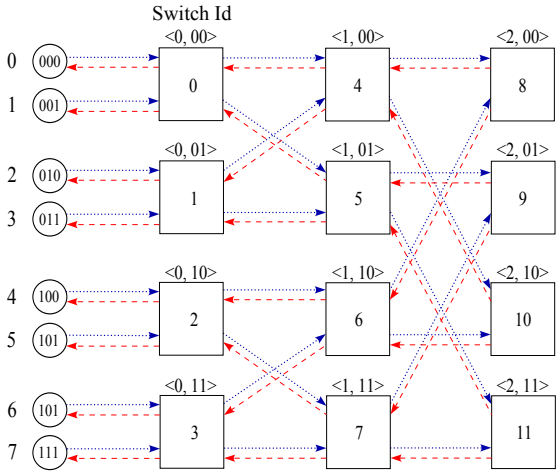
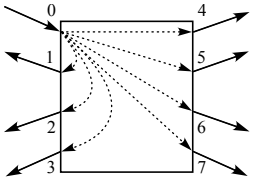


Fig. 1: A 2-ary 3-tree interconnection network.

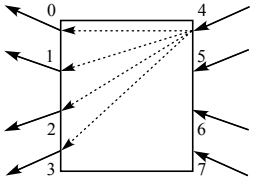
A. Adaptive Routing in Fat–trees

In k–ary n–trees, minimal routing from a source to a destination can be accomplished by sending packets upwards to one of the nearest common ancestors of the source and destination nodes and then, from there, downwards to destination. When crossing stages in the upwards direction, several paths are possible, thus providing adaptive routing. In fact, each switch can select any of its k up output ports. Once one of the nearest common ancestor has been reached, the packet is turned around and sent downwards to its destination as just a single path is available. The stage up to which the packet must be forwarded is obtained by comparing the source and destination components beginning from the most significant one. The first pair of components that differs indicates the last stage to forward up the packet. Once in that stage, the descending path is deterministic. At each stage, the descending link to choose is indicated by the component corresponding to that stage in the destination n–tuple.

Given that the routing algorithm is adaptive, in the upwards subpath, at each switch, the k input ports can forward packets through either any of the up k output ports, if the packet continues in its upwards subpath, or any of its down output ports if the packet starts its downwards subpath. On the other hand, in the downwards subpath, there are k down input ports that can only request k down output ports, since once a packet has started its downwards subpath, the packet must continue going downwards. Figures 2a and 2b show the output ports that can be requested in the upwards and downwards directions, respectively, in the switches of a 4–ary n–tree.



(a) Requested ports in the upwards direction by port 0.



(b) Requested ports in the downwards direction by port 4.

Fig. 2: Ports that can be requested in a 4–ary n–tree using adaptive routing.

Figure 3 depicts an example of a packet that is sent from processing node 0 to processing node 7. As can be seen, in the upward phase, the packet can take any output port thus providing a high number of possible paths (dotted lines), meanwhile at the downward phase the packet only has one possible path that depends on the chosen path at the upwards phase (dashed lines). The number of paths tends to exponentially increase as the network size increases.

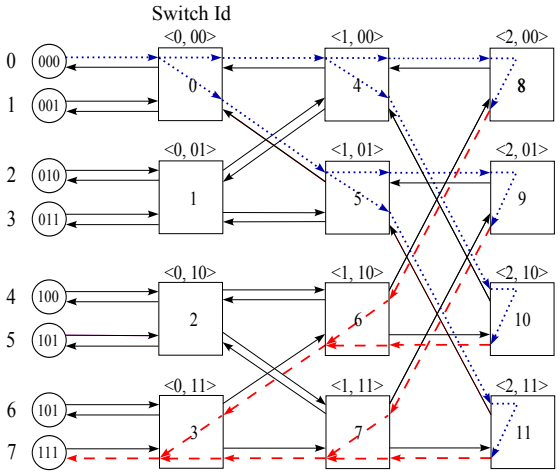


Fig. 3: Adaptive Routing.

B. Fault-Tolerance in Fat-Trees

In interconnection networks, fault-tolerance is usually implemented by using a routing mechanism that provides several

alternative paths to communicate every source–destination pair. To reach this goal, some proposals add links and switches to the base network.

However, in the case of fat–trees, as stated above, adaptive routing already provides several disjoint paths to send information from each processing node to the corresponding destination. This flexibility might not be only used for performance-improvement purposes but also for fault-tolerance. In particular, the number of alternative paths between each source–destination pair increases with the network arity (k, related to switch degree) and the number of stages n. Depending on the destination, there are a maximum of $k^{(n-1)}$ different paths.

III. Parallel computing

Parallel computing is a form of computation in which many calculations are carried out simultaneously to solve problems that require either large processing time or handling large amounts of data in the shortest possible time. The parallel processing philosophy is to divide the problem into simple tasks and solve them concurrently. However, we must keep in mind that not all problems can be parallelized.

Among the most common forms of parallelizing an application are MPI, OpenMP and, using a GPU. The main drawback of MPI and OpenMP is that the number of parallel tasks is limited by the number of processing elements, which is in the order of tens for most used installations. On the other hand, GPUs offer a large processing capacity in a single unit of hardware at a very low cost. For this reason, most parallel applications are being ported to GPUs, as their use represents economic savings in several aspects (acquisition, maintenance, power consumption, etc.) while providing high performance.

A. CUDA

The first GPUs were designed as graphics accelerators. From there their architecture evolved from a specific single-core with a fixed-function hardware pipeline made solely for graphics, to a set of highly parallel and programmable cores for more general purpose computation. Starting in the late 1990s, the hardware became increasingly programmable, culminating in NVIDIA's first GPU in 1999. From 2001 to 2005 the evolution of CPUs and GPUs was similar. Since 2006, GPU performance increased significantly. In 2009, the peak floating–point calculation throughput ratio between CPUs and GPUs was about 10 to 1, which means GPUs reached 1 teraflop and CPUs only 100 gigaflops [11].

The idea of using GPUs for intense numeric computing motivated the design of CUDA (Compute Unified Device Architecture), a programming model for execution of an application to take advantage of GPUs [11].

The architecture of CUDA (Figure 4) is organized in modules called Streaming Multi–processors (SMs). Two SMs form a block (however, the number of SMs in a building block can vary from one generation of GPUs to another) each one with an independent parallel cache, and a Global Memory. The SMs are composed of Streaming Processors (SPs) that share a control logic and an instruction cache. The total number of SPs depends on the GPUs model. The SPs can execute multiple threads per application (even thousands of threads). Meanwhile, CPUs only support 2 or 4 threads per core, according to the current trends.

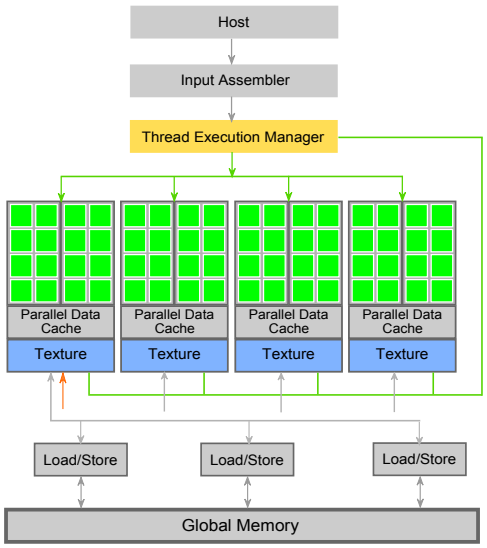


Fig. 4: Architecture of a CUDA–capable GPU.

Parallel programming in CUDA is explicit and fine grain, i.e., it is necessary to design how the task will be divided to be executed in parallel and how the communication between tasks can be performed. This architecture is composed by several elements, and to understand a CUDA code, it is necessary to know some basic concepts about it, such as: *host, device, function qualifier, kernel, global memory, constant memory, block* and, *grid*.

- *Host* and *device* are used to refer to the hardware. Host is used for the CPU side of the system and device for its GPU counterpart.
- The *function qualifiers* are used to indicate where the function will be executed and its type of access. The used qualifiers are *__host__*, *__device__*, and *__global__*. The first one refers to the function that is callable and executed only in the CPU, while the second one will run on a GPU and not on the host. Besides, this function will be callable only from other *__device__* functions or from *__global__* functions. The latter means that the function serves as the point of entry into a *kernel* which executes in parallel on a GPU device.
- A *kernel* is a function executed on the GPU as an array of threads in parallel. All threads execute the same code and can take different paths. The kernel always is marked as __global__.
- The *Global Memory* is tipically implemented in DRAM. It is analogous to RAM in a computer and can be accessed by both GPU and CPU. It is accessible by all threads of any kernel and its datalife time is from allocation to deallocation by host code. The latency of this memory is between 400 and 800 cycles. Currently, we can found

some GPUs with up to 6GB of memory.

- The *Constant Memory* is used for data that will not change over the course of a kernel execution, and supports short-latency, high bandwidth, read-only access by the device when all threads simultaneously access the same location.
- A *block* is a set of threads. The block size or maximum number of threads per block depends on the GPU model.
- A *grid* is a collection of blocks. A grid can be one, two, or tri-dimensional. As the block size, the grid size is limited by the GPU implementation.

IV. The problem

The fault-tolerance analysis is based on taking a network and introduce a fault or a set of faults in the interconnection links and test the connectivity of the system, checking that each processing node has, at least, one route to all destinations (except itself). Once finished this checking, the faults are moved to other links and the checking process is repeated again. If, for example, we have a network with 32 unidirectional links and we are testing a single fault, the process will take place 32 times. In Table I, we can see how increasing the number of faulty links, in a small 2–ary 3–tree network using a fat–tree with adaptive routing, leads to an exponential increase in the number of combinations to evaluate.

Number of faults to evaluate	Combinations of faults
1	32
2	496
3	4960
4	35960
5	201376
10	64512240
15	565722720
16	601080390

TABLE I: Number of combinations of faults for a 2–ary 3–tree network, considering up to 16 faults.

If, for a given number of faults, for all the possible combinations of this number of faults, the topology is able to provide at least a path for every source–destination pair, then the topology tolerates that number of faults. If some of the combinations are tolerated and others are not, then that number of faults is not tolerated, but in this case, we can obtain the percentage of combinations that are tolerated, that is, those that have at least a healthy path for each source-destination pair.

This type of analysis has a major drawback: the processing time. Two reasons impact the execution time: *1*) as the number of faults to evaluate increases, so does the number of combinations that must be evaluated, and *2*) as the network size increases, so does the number of network links. In addition, some network topologies use bidirectional links by design, as the fat–trees. In these cases, it is necessary to consider these links as unidirectional to perform a deeper analysis, so every switch with an arity of k, has $2k$ input ports and $2k$ output ports (see Figure 1). Thus, if an up link fails, it does not affect the corresponding downlink. So when considering unidirectional links, the number of links (and locations for faults) to evaluate will double.

Those who have addressed this issue have developed tools working on a single CPU, obtaining a low execution performance. As we mentioned before, the major drawback of this analysis is the high execution time, because evaluating a single combination of faults is a process that may be too long, since the complexity of the algorithm is in order of:

$$N * (N-1) * Max\_Nr\_Paths * Nr\_Combinations$$

Where the N is the number of processing nodes in the network. In a k–ary n–tree network, the number of nodes is equal to k^n. Therefore, the term $N * (N-1)$ indicates that all the routes between all nodes must be evaluated.

The maximum number of paths (Max_Nr_Paths) that exist in a network, depends on the topology that is being evaluated. For the case of a fat–tree with adaptive routing, this value is given by the expression:

$$Max\_Nr\_Paths = N * \sum_{i=k}^{N-1} k^{\lfloor \log_k i \rfloor}$$

Finally, the number of combinations ($Nr\_Combinations$) that can be obtained for a given number of faults is given by the formula:

$$Nr\_Combinations = \frac{Network\_Links!}{fte!(Network\_Links - fte)!}$$

The variable $Network\_Links$ is the number of links that exist in the network for a fat–tree with adaptive routing, and can be calculated using the formula:

$$Network\_Links = \frac{k^n}{k} * (n-1) * k * 2$$

The variable fte is the number of faults to be evaluated when performing the network analysis.

The high complexity of the algorithm makes infeasible the analysis of large networks on a CPU. Although statistically is not necessary to evaluate all the faults, it is very important to analyze a representative sample. However, in medium and large-sized networks performing this analysis on a single CPU is not feasible, because the processing time for a single combination of faults will be prohibitive. Such applications may be parallelized in large machines, such as a cluster, but the number of cores is much lower than that available in a single GPU. Alternatively, multiple GPUs or even an hybrid (CPU-GPU) version could be used.

V. The algorithm

Following the philosophy of parallelization, we have divided our problem into smaller parts, so that each thread on the GPU is responsible of analyzing a part of it.

As stated before, the fault-tolerance analysis is based on testing thousands of combinations of faults in the network links, and in turn, for each combination of faults, the connectivity among all processing nodes of the network must be checked.

From our point of view, we could tackle this analysis in two different ways: *1*) evaluate several combinations of faults in the GPU at the same time or *2*) to use the GPU evaluating only one fault combination at a time.

In the former, we could copy several network instances to the GPU and assign each one of them to a GPU thread (one fault combination per thread). However, although each combination of faults to evaluate is independent of the rest, it is not appropriate to assign long processes to each thread since the processing time for a single analysis can take considerable time. Furthermore, with this approach, we have to keep in mind two important aspects: *1*) different fault combinations can take different processing time, given that the fault location affects the number of paths to evaluate. So, if we assign a combination of faults to each thread we would find a barrier, and would have to wait for the last thread to continue the analysis. *2*) as the network size increases, the number of instances that fit in the GPU is lower, due to limitation of memory in the device.

In the latter case, to speed up the application execution, we assign only a fault combination to the GPU, i.e. copying only a network instance to the GPU, so that each thread will check the connectivity between a source-destination pair. Thus, as the network grows so does the level of parallelization, allowing us to obtain better execution times on the GPU.

This is the approach followed in this paper. The Algorithm 1 shows the general steps that allows us to perform the required analysis.

input : Network and interconnection topology.
output: Fault-tolerance stats.

Determinate the grid size, according to the network size and GPU capability;

for $i \leftarrow 0$ **to** $Nr\_Combinations - 1$ **do**
  $\Rightarrow$ Generate a single combination of faults;
  $\Rightarrow$ Apply the combination of faults to the network;
  $\Rightarrow$ Copy the network and auxiliar elements to device;
  $\Rightarrow$ Launch `Kernel`;
  $\Rightarrow$ Copy results from device to host;
  $\Rightarrow$ Collect processed data;
  $\Rightarrow$ Free memory;
  $\Rightarrow$ Remove faults from network;
end

Algorithm 1: Algorithm for analyzing fault-tolerance.

At the GPU side, to avoid overloading the kernel with unnecessary loops or using arrays to store the source-destination pair to evaluate, these values are calculated using the thread identifier, as is shown in Algorithm 2. For the best performance, it would be interesting to copy the network to the constant memory of device, but since the size (in bytes) of the network is generally higher than the constant memory space (16KB-64KB in most cases), we are forced to use the global memory.

input : Network with faults.
output: Fault-tolerance stats.

Calculate the thread identifier (tid);

//Compute the origin–destination pair to evaluate;
orig = tid / (N-1);
dest = (tid + orig + 1) mod (N-1);

if (orig == dest)
  dest=N-1;
end

stats[tid]=EvaluatePaths(orig, dest);

Algorithm 2: Kernel code.

The *EvaluatePaths* function checks if there is an available path between the given source–destination pair. A given combination of faults is tolerated if the result of this function is *true* for all source–destination pairs. An in-depth description of this function is out of the scope of this paper.

VI. Evaluation

In this section, we will evaluate the fault-tolerance of a fat tree with adaptive routing to analyze the performance of the tool running on a GPU, comparing the results with the same tool running on the CPU.

A. Simulation Enviroment

To obtain the results of this section, we have developed two versions of the fault-tolerance analyzer, one for each type of hardware. The CPU version runs on a Intel Xeon E5530, while the GPU version runs on a Nvidia Tesla C1060 card.

Tables II and III show the most relevant specifications of the used CPU and GPU, respectively, to carry out the present work.

Model name	Intel®Xeon®
CPU	E5530 @ 2.40GHz
Cache size	8 MB
CPU cores	4

TABLE II: CPU specifications.

Model name	Tesla C1060
Compute capability	1.3
Clock rate	1.296GHz
Total global memory	4GB
Total constant memory	64KB
Multiprocessor count	30
Shared memory per block	16KB
Register per block	16384
Max threads per block	512
Max thread dimensions	[512 512 64]
Max grid dimensions	[65535 65535 1]

TABLE III: GPU specifications.

The topology has been analyzed for different network sizes, such as a 4–ary 3–tree, a 8–ary 3–tree and, 16–ary 3–tree. To prevent that the processing time on the CPU becomes very high, we bounded the number of tested combinations to 100000 when the combinations of faults were higher to this value. In both versions of the tool, for timing analysis purposes, we only have considered the function that analyzes the fault-tolerance; the fault combinations generator is not taken into account because it does not represent a processing overhead compared to the execution of the function/kernel that perform the fault-tolerance analysis.

B. Evaluation Results

As stated above, we will show the results for both CPU and GPU processing time. To better observe the differences between the two approaches, the execution time in the figures have been represented in logarithmic scale.

In Figure 5, we can see the processing time for a 4–ary 3–tree, the smallest network analized, using both versions of the tool. As can be seen, both execution times are very similar. This is because, since the network size is very small (64 processing nodes), the number of network links is also small. For this reason, the level of exploited parallelism in the GPU is low, getting a performance close to the CPU.

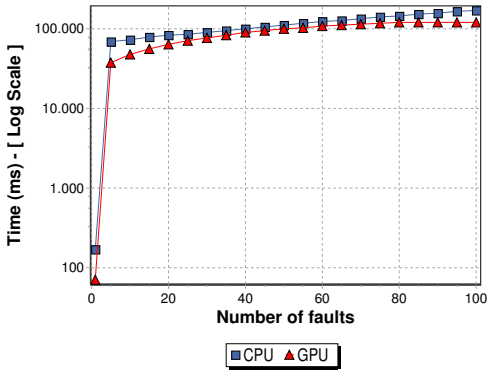


Fig. 5: Execution time of the tools for a 4–ary 3–tree.

For a 8–ary 3–tree network, as it can be seen in Figure 6, the differences in processing time between both versions of the analyzer are larger because the number of source–destination pairs that must be evaluated is higher, so the potential of the GPU can be better exploited. For this network size the GPU–based tool has performed the analysis 23X faster than the CPU. In other words, while the GPU performs the analysis in 3.8 minutes, for 100 faults, the CPU does the same work in 1.5 hours.

As expected, by analyzing larger networks we can see that the processing time increases significantly. Figure 7 shows results for a 16–ary 3–tree. In this case, evaluating 5 faults takes 67.8 hours in the CPU while its counterpart in the GPU, performs the analysis in only 2.3 hours. The speedup of the process was more than 27X.

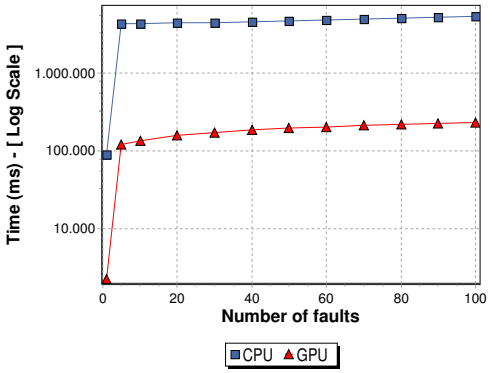


Fig. 6: Execution time of the tools for a 8–ary 3–tree.

As we can see, the CPU–based tool can not be used in practice for medium to large networks. Table IV shows the number of network links and the maximum number of paths that must be evaluated for the fat–tree topology (with adaptive routing) using different network sizes and 100000 combinations of faults. As it can be seen, this number exponentially grows with the network size and also the CPU execution time. The GPU–based tool effectively enables analyzing these network sizes in a reasonably execution time.

As an example of usage of the developed tool, following we show some of the obtained results for fault-tolerance. In Figure 8 we can see the number of source–destination pairs that are able to communicate for different number of faults and two network sizes. In both cases, we have evaluated up to 256 faults in the network. As we can see, although the exhibited networks have the same number of processing nodes, the number of network links is higher for the 2–ary 6–tree network; therefore, for the same number of faults, its impact is lower. The number of paths is greater and, in turn, better exploited by the adaptivity of the routing algorithm. Another important result is shown in Figure 9. Here we can see the percentage of non tolerated fault combinations, i.e., of the 100000 tested combinations, it shows what percentage of them fails to establish a path between all origin-destination pairs. In this case, for the 4–ary 3–tree network, the number of tolerated faults is higher than for the another network, i.e. this network supports more faults. This is because the arity of the used switches in the network is higher, therefore the number of connections among switches will also increase, allowing us to reach a higher degree of fault-tolerance.

It is noteworthy that the developed tool supports using multi-GPU, which provides a higher speedup in case of being required. Moreover, it is also possible to run several instances of the GPU–based tool on different nodes of a parallel machine (i.e. a cluster of computers) provided that every one has a GPU installed. Both ways of exploiting parallelism (i.e. at each node and at each GPU) will lead to improved results.

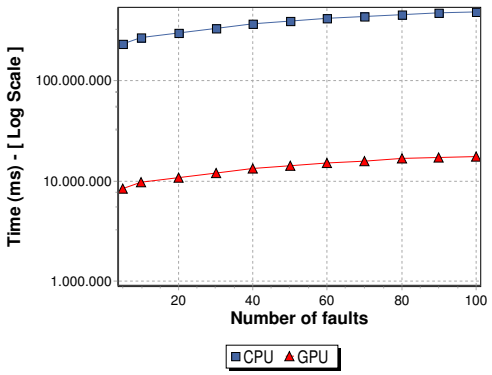


Fig. 7: Execution time of the tools for a 16–ary 3–tree.

Network Size	Processing Nodes	Network Links	Paths to Evaluate
2–ary 6–tree	64	640	8,72960E+09
4–ary 3–tree	64	256	5,22240E+09
4–ary 5–tree	1024	8192	2,14745E+13
8–ary 3–tree	512	2048	1,49094E+12
8–ary 5–tree	32768	262144	3,90937E+17
16–ary 3–tree	4096	16384	4,04226E+14

TABLE IV: Number of paths to evaluate depending on the network size.

VII. Conclusions

We have presented an approach to optimize the fault-tolerance analysis of multistage interconnection networks using parallelism. In particular, we take advantage of the GPU device available in most current systems. The fault-tolerance analyzer based on a single CPU have a very high processing time. Although several instances of the tool can be running on a cluster of computers, it results in a high power consumption and also the number of processing cores is bounded in practice to tens of nodes. By using the GPU–based version, we have obtained an improvement of up to 23X compared with the CPU counterpart, allowing us to perform a more complex analysis in a finite and reasonably execution time.

Furthermore, although this work has focused in the fat-tree topology, the fault-tolerance analysis, and consequently its optimization, can be used on other direct or indirect interconnection networks.

Acknowledgment

This work was supported by the Spanish Ministerio de Economa y Competitividad (MINECO) and by FEDER funds under Grant TIN2012-38341-C04-01.

References

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

[2] S. Bataineh and B. Allosl. Fault-Tolerant Multistage Interconnection Network. *Telecommunication Systems*, 17(4):455–472, 2001.

[3] C. Chen and C. Chung. Designing a disjoint paths interconnection network with fault tolerance and collision solving. *The Journal of Supercomputing*, 34(1):63–80, 2005.

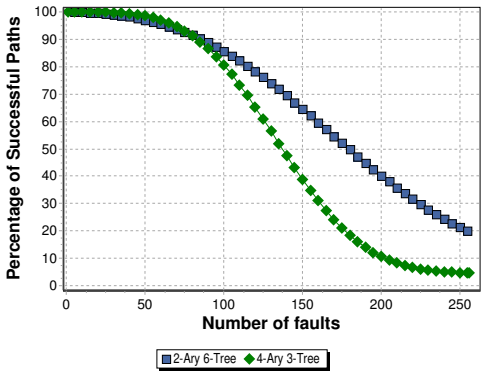


Fig. 8: Percentage of successful source–destination paths for different network sizes.

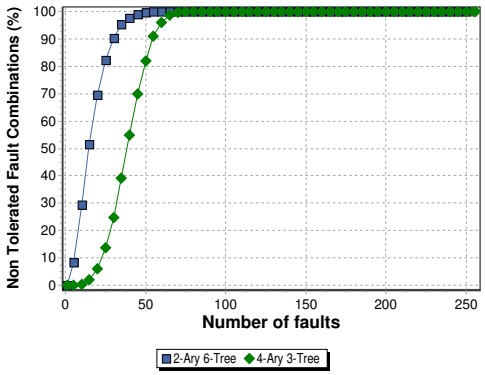


Fig. 9: Percentage of non tolerated faults for different networks sizes.

[4] C. Chen, P. Gan, and C. Chang. Designing a High Performance and Fault Tolerant Multistage Interconnection Network with Easy Dynamic Rerouting. In J. Cao, L. Yang, M. Guo, and F. Lau, editors, *Parallel and Distributed Processing and Applications*, volume 3358 of *Lecture Notes in Computer Science*, pages 1007–1016. Springer Berlin Heidelberg, 2005.

[5] C. Chen, N. P. Lu, T. Chen, and C. Chung. Fault-tolerant gamma interconnection networks by chaining. *Computers and Digital Techniques, IEE Proceedings* -, 147(2):75–81, March 2000.

[6] F. Chong and T. Knight, Jr. Design and Performance of Multipath MIN Architectures. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, pages 286–295, New York, NY, USA, 1992. ACM.

[7] P. Chuang. CGIN: a fault tolerant modified Gamma interconnection network. *Parallel and Distributed Systems, IEEE Transactions on*, 7(12):1301–1306, December 1996.

[8] I. Gazit and M. Malek. Fault tolerance capabilities in multistage network-based multicomputer systems. *Computers, IEEE Transactions on*, 37(7):788–798, July 1988.

[9] C. Gómez, M. Gómez, P. López, and J. Duato. An efficient fault-tolerant routing methodology for fat-tree interconnection networks. In *Fifth International Symposium on Parallel and Distributed Processing and Applications (ISPA07)*, pages 509–522, August 2007.

[10] N. Kamiura, T. Kodera, and N. Matsui. Design of a fault tolerant multistage interconnection network with parallel duplicated switches. In *Defect and Fault Tolerance in VLSI Systems, 2000. Proceedings. IEEE International Symposium on*, pages 143–151, 2000.

[11] D. B. Kirk and W. mei W. Hwu. *Programming Massively parallel Processors*. Morgan Kaufmann, 2010.

[12] D. S. Parker and C. Raghavendra. The Gamma Network. *Computers, IEEE Transactions on*, C-33(4):367–373, April 1984.

[13] F. Petrini and M. Vanneschi. k-ary n-trees: high performance networks for massively parallel architectures. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 87–93, April 1997.

[14] R. Rastogi, Nitin, and D. Chauhan. 3-Disjoint Paths Fault-tolerant Omega Multi-stage Interconnection Network with Reachable Sets and Coloring Scheme. In *Computer Modelling and Simulation (UKSim), 2011 UkSim 13th International Conference on*, pages 551–556, March 2011.

[15] R. Rastogi, R. Verma, Nitin, and D. Chauhan. 3-Disjoint Paths Fault-tolerant Multi-stage Interconnection Networks. In A. Abraham, J. Lloret Mauri, J. Buford, J. Suzuki, and S. Thampi, editors, *Advances in Computing and Communications*, volume 190 of *Communications in Computer and Information Science*, pages 21–33. Springer Berlin Heidelberg, 2011.

[16] N. Sharma. Fault-tolerance of a min using hybrid redundancy. *in Proc. of the 27th Annual Simulation Symp*, 1994.

[17] M. Valerio, L. Moser, and P. Melliar-Smith. Fault-tolerant orthogonal fat-trees as interconnection networks. In *Algorithms and Architectures for Parallel Processing, 1995. ICAPP 95. IEEE First ICA/sup 3/PP., IEEE First International Conference on*, volume 2, pages 749–754, April 1995.

[18] G. Zarza, D. Lugones, D. Franco, and E. Luque. A Multipath Fault-Tolerant Routing Method for High-Speed Interconnection Networks. In H. Sips, D. Epema, and H.-X. Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 1078–1088. Springer Berlin Heidelberg, 2009.