# Efficiency and Scalability of Barrier Synchronization on NoC Based Many–core Architectures

Oreste Villa
Pacific Northwest National Laboratory,
High Performance Computing,
902 Battelle Boulevard,
99354 Richland (WA), USA
oreste.villa@pnl.gov

Gianluca Palermo
Politecnico di Milano,
Dipartimento di Elettronica e Informazione,
Via Ponzio 34/5,
20133 Milano, Italy
gpalermo@elet.polimi.it

Cristina Silvano
Politecnico di Milano,
Dipartimento di Elettronica e Informazione,
Via Ponzio 34/5,
20133 Milano, Italy
silvano@elet.polimi.it

## ABSTRACT

*Interconnects based on Networks-on-Chip are an appealing solution to address future microprocessor designs where, very likely, hundreds of cores will be connected on a single chip. A fundamental role in highly parallelized applications running on many-core architectures will be played by barrier primitives used to synchronize the execution of parallel processes. This paper focuses on the analysis of the efficiency and scalability of different barrier implementations in many-core architectures based on NoCs. Several message passing barrier implementations based on four algorithms (all-to-all, master-slave, butterfly and tree) have been implemented and evaluated for a single-chip target architecture composed of a variable number of cores (from 4 to 128) and different network topologies (mesh, torus, ring, clustered-ring and fat-tree). Using a cycle-accurate simulator, we show the scalability of each barrier for every NoC topology, analyzing and comparing theoretical with real behaviors. We observed that some barrier algorithms, when implemented in hardware or software, show a different scaling behavior with respect to those theoretically expected. We evaluate the efficiency of each combination topology-barrier, demonstrating that, in many cases, simple network topologies can be more efficient than complex and highly connected topologies.*

**Categories and Subject Descriptors:** C.4 Computer Systems Organization Performance of Systems

**General Terms:** Algorithms, Design, Performance

**Keywords:** Barrier, Efficiency, Manycore, Multicore, NoC, Scalability, Synchronization

## 1. INTRODUCTION

Many-core architectures based on NoCs [3, 6] are a promising solution to tackle modern and future processor design challenges, as power consumption, scalability and ease of design reuse. The basic idea is to use on-chip packet switches as building blocks to create a micro network as interconnect, similarly to what appends in traditional large-scale multiprocessors and distributed computing networks. NoC interconnects provide a low latency communication layer which solves physical limitations due to wire latency, providing when compared to bus based solutions higher bandwidth and parallelism in the communication. Great attention has been posed by many researches to find clever topologies and optimized routing algorithms able to meet such expectations [15, 12, 14].

Current NoC based architectures are still in their infancy and, only few companies (mainly not mainstream oriented) are starting to develop and to commercialize them, in specific domains such as transaction intensive applications and network processing [17, 5]. This is mainly due to the fact that current manufacturing technologies do not allow to integrate more than a dozen of cores of medium complexity (i.e. simple pipelines without Out-of-Order capabilities) while the NoC approach gains in performance and power (over traditional bus based solutions) when the number of interconnected units become relevant. However, in the near future, as predicted by many [4, 2], as CMOS technologies will continue to shrink and the number cores integrated on a chip will increase, NoC based interconnects will start – hopefully– to reach mainstream products. A recent example of this trend has been shown with the *Polaris* test chip in the *Tera-Scale* [8] research initiative proposed by Intel where eighty simple cores are connected by using a bidirectional mesh. This chip, while still unable to execute complex control structures, is capable to deliver teraflop-level performance consuming less than 100 Watts.

One of the key points in multi-processor programming is that, when multiple processing elements are concurrently executing the same application, it is necessary to ensure correctness by adding synchronization points. Compiler-parallelized and scientific data-intensive applications use collective communication primitives to synchronize parallel execution but, as the number of cores increases, their implementation becomes a challenging task. Conventional techniques require to insert a *barrier* [18] in a point of the code where we want that all processes (or a given number of them) arrive before the elaboration proceeds. Barriers are latency sensitive global operations which can be realized with many different techniques (centralized counters, all-to-all global exchange, etc.) and adopting several algorithms (master slave, butterfly, tree, etc.). Barriers require low latency communication channels to minimize overall completion time and due to their global nature, if not carefully implemented, can require thousands of cycles to be completed when hundreds of processes are involved.

NoC interconnects seem to meet the demand of low latency, because of the parallel nature of the communication, but how quantitatively behave typical barrier algorithms in NoC architectures when up to hundreds of cores are involved? This paper tries to answer to this question by proposing a quantitative analysis (based on a cycle-accurate many-core simulator) to understand how different NoC topologies behave when dealing with different SW/HW barrier implementations.

To achieve this result, we considered a target architecture composed of a variable number of cores (from 4 to 128) interconnected by five different network topologies (mesh, torus, ring, clustered-ring, and fat-tree). Then, we present four barrier algorithms (all to all, master slave, butterfly and tree [18]), and we show their SW and HW implementations. In our analysis, we focus on message passing based barriers neglecting barrier implementations that involve the main memory (i.e. centralized counters barriers) since they have already been demonstrated to be less efficient for on-chip synchronization [10].

In this paper, we investigate the behavior of hardware and software barrier synchronization in the most likely future many-core architectures, when several on-chip network topologies are taken into account. More in detail, the main contribution of this paper is to analyze the *scalability* of different barriers over a set of selected NoC topologies by using a cycle-accurate simulator. Simulated results have been compared with theoretical behaviors of barrier performance. The theoretically expected behavior of barriers (such as the linear or logarithmic dependency of the latency with respect to the number of processes) has been confirmed by our simulation results. The scaling behavior among barriers has been quantified and, in particular, we observed that this behavior is different when barriers are implemented in hardware or in software and when different network topologies are considered. This result makes it difficult to identify the best barrier implementation for the different scenarios, therefore justifying our trade-off analysis.

Another goal of our analysis is to evaluate, for barrier synchronization, the overhead of the different NoC topologies with respect to an ideal channel when scaling the number of processors in the system. One of the most significant results of this analysis is that, given the high latency of message passing SW barriers, the impact of NoC topology is not relevant so as the usage of highly connected network topology is not fully justified in these cases. On the other side, we proved that NoC topology has a more relevant impact when dealing with HW barriers.

Finally, we also evaluated the *efficiency* of each combination topology/barrier by trading-off the complexity cost for each NoC configuration with the latency of the barrier operations. We experimentally demonstrated that complex topologies behave similarly (in terms of latency) to simpler topologies for several barriers algorithms.

The paper is organized as follows. Section 2 describes the related works. Section 3 shows the target architecture and the network topologies that we consider in our analysis. Section 4 describes the different barrier algorithms as well as their SW and HW implementations. Section 5 describes the experimental setup. Section 6 shows and comments the results of our experiments, while Section 7 concludes the paper and points out our future research.

## 2. RELATED WORK

Traditionally, many researches in the field of High Performance Computing (HPC) have extensively studied the relation between network topologies and Collective Communication Operations (CCOs) (such as barriers) in the form of algorithms and their implementations. In [16], the authors analyzed how HW and SW based barriers are designed and implemented in a programmable Network Interface Card (NIC) and demonstrated that without network contention the SW and HW approach can be used interchangeably on a fat-tree topology for systems of 64-128 nodes. Yu et al. [19] demonstrated that most of the communication latency due to barriers can be reduced with the use of a collective HW protocol located in the NIC of the different nodes in large scale systems. In the field of chip multiprocessors, while Liu et al. [9] demonstrated the importance and the impact (in terms of performance overhead and power consumption) of barrier synchronization, in [10] the authors discuss the use of a run-time lightweight barrier construct in non cache coherent MPSoC platforms. Finally, in [11] the authors propose a centralized HW solution to perform barrier synchronization in embedded systems composed up to 8 cores.

This work is a further step in the study of barrier synchronization: it finds inspiration from the above works and it tries to address, in the single-chip multiprocessors field, some issues that are typical of the HPC community. In many aspects, as the number of cores integrated on a single chip increases, the similarities with parallel computers must be further investigated especially regarding collective communication operations such as barriers.

## 3. TARGET ARCHITECTURE

In this Section, we describe what we believe could be a future many-core architecture, when potentially hundreds of cores are integrated on a single chip. Our target architecture is very similar to the one envisaged by Intel in their Tera-scale computing research program [8]. Fig. 1 shows our target platform as composed of $N$ simple cores (CRs) (with their local and global caches) and $M$ external memory interfaces (EMIs) which are the entry/exit point to the off-chip main memory. All the units are connected together with a highly scalable NoC interconnect. Although a more general architecture could contain specialized hardware modules we do not consider them for the purpose of our analysis.

As pointed out in [4, 2], most of the complexity that we see in today microprocessors (e.g. deep pipeline, out-of-order execution, register renaming, dynamic schedulers, complex branch prediction, reorder buffers) will be partially reduced or perhaps removed when moving to many-core architectures. Under this assumption, we propose our many-core building block as the PowerPC 750 [7] architecture. The PowerPC 750 is a simple out-of-order single-thread processor capable to issue and to complete up to two instructions per clock cycle. Its transistor count (without taking into account the L2 unified cache) of 4.5 Millions makes it a suitable candidate for future multi-billion transistors chips, where hundreds of such cores can be integrated. However, the results that we present in this paper are qualitatively independent from the selected core's architecture and are

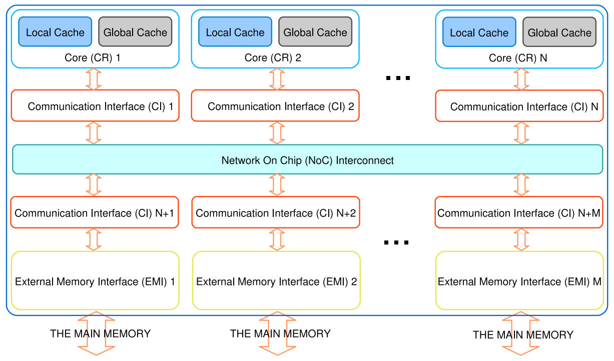

Local Cache
Global Cache
Core (CR) 1
Core (CR) 2
Core (CR) N
Communication Interface (CI) 1
Communication Interface (CI) 2
Communication Interface (CI) N
Network On Chip (NoC) Interconnect
Communication Interface (CI) N+1
Communication Interface (CI) N+2
Communication Interface (CI) N+M
External Memory Interface (EMI) 1
External Memory Interface (EMI) 2
External Memory Interface (EMI) M
THE MAIN MEMORY

**Figure 1: The target many-core architecture**

representative of a more general class of architectures with a complexity comparable to that of the PowerPC 750.

Each PowerPC 750 core is provided with a communication interface (CI) which allows: (I) to pack and unpack information received and injected in the NoC, (II) to manage message passing communication layer and (III) to embed the HW implementations of the barrier algorithms that we will show later. The CI is a memory mapped interface device with I/O buffers where commands are enqueued and processed. The interface is able to send and receive up to 8 bytes every clock cycle. Its message passing interface is accessible directly from the application layer through a non-blocking send operation (i.e.*send(data, NoC_address)*) and a blocking receive operation (i.e. $data = recv()$) which returns the first value in the CI's input buffer.

In order to connect the different modules (CRs and EMIs) we developed five different NoC based topologies (mesh, torus, ring, clustered-ring and fat-tree) where each building block is a router with a variable number of bidirectional ports (from 2 to 5). Fig. 2(a) shows a schematic of the router's architecture as well as the number of equivalent gates [1] for each configuration. This value is linear with the number of ports, since area is dominated by I/O buffers. The router is a 3 stage pipeline, with I/O ports of 32 bit. The I/O buffers are 16 byte each, the internal switch is implemented with a small crossbar and the routing arbitration logic algorithm is static table based. In our experiments, we also considered a crossbar topology with zero delay, namely *ideal channel*. This, more than a possible real configuration, represent a reference when comparing to switch based topologies; indeed, due to its point-to-point nature, the zero delay crossbar gives us the reference latency to compare switch-based NoCs. Fig. 2 shows the organization of the five different NoC topologies. The clustered-ring topology is realized with rings of size 8 (therefore clustered-ring and ring are equivalent for topologies with less than 8 modules). The packet injection and ejection ports (I/E ports) are represented as bidirectional 45° arrows. The fat-tree topology is the only one to have routers with two I/E ports (see Fig. 2(f)).

We explored each topology connecting from 4 to 128 CRs and from 2 to 7 EMIs. We define a complexity cost "C" for each topology which will be used to find its efficiency while performing barrier operations. The complexity cost is obtained as the sum of the number of ports ($J_i$) for each of the $L$ routers in the topology ($C = \sum_{i=1}^{L} J_i$). Since the number of equivalent gates is linearly dependent on the number of ports (see Fig. 2(a)), the cost "C" is proportional to (i) the overall size of the NoC (ii) the complexity and length of the interconnects between routers in the topology (iii) the leakage power lost in the NoC. Therefore we can assume "C" as a good estimator of the relative complexity of the different NoCs. Table 1 shows the cost "C" for each topology for different sizes of the system.

[1]Synthesized with Synopsys and STM 90nm libraries from a VHDL description.

**Table 1: Complexity cost of the five different topologies as function of the system size. (CRs = number of cores, EMIs = External Memory Interfaces)**

| System Size | Network Cost | | | | |
|---|---|---|---|---|---|
| | | | | Clustered | |
| CRs+EMIs | Mesh | Torus | Ring | Ring (size 8) | Fat Tree |
| 4+2 | 30 | 20 | 18 | 18 | 25 |
| 8+3 | 55 | 41 | 33 | 31 | 65 |
| 16+4 | 100 | 82 | 60 | 66 | 153 |
| 32+5 | 185 | 160 | 111 | 121 | 348 |
| 64+6 | 350 | 316 | 210 | 228 | 788 |
| 128+7 | 675 | 626 | 405 | 439 | 1776 |

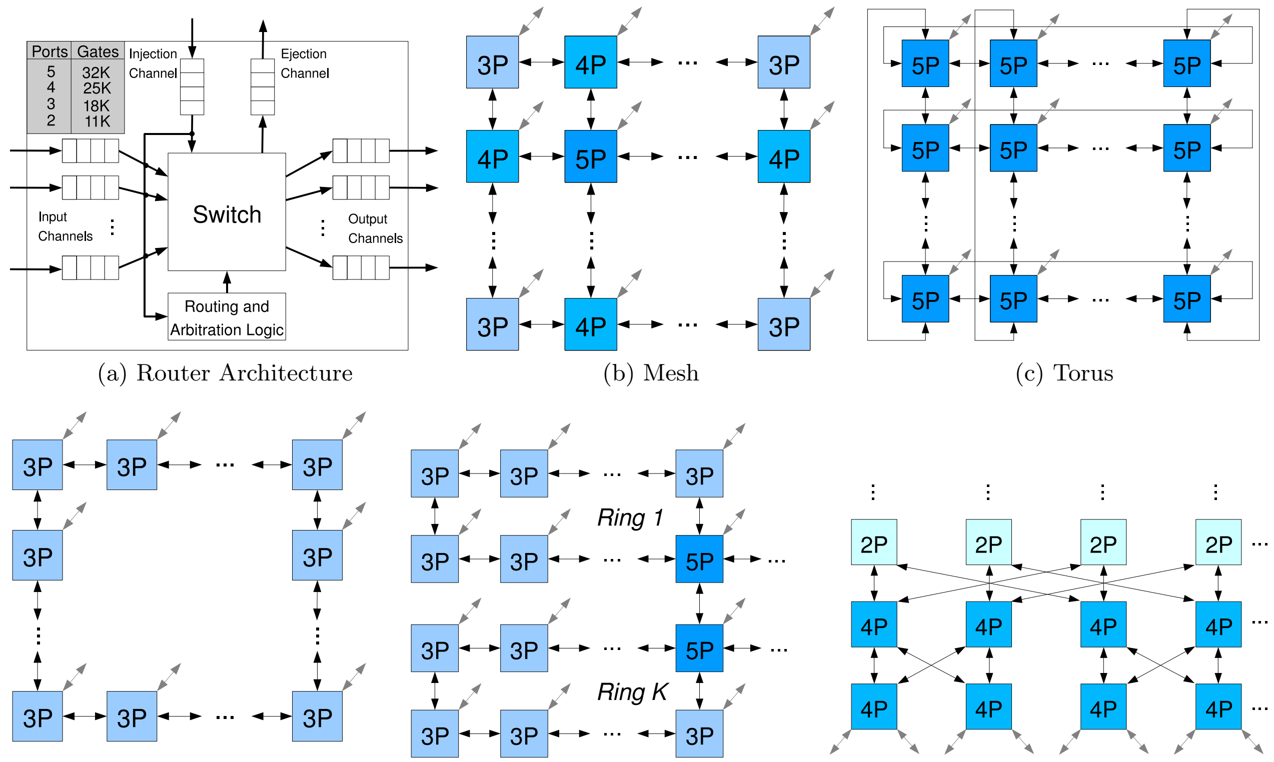
## 4. BARRIER ALGORITHMS

The cores of our system can execute only one process at time, therefore in the rest of the paper we consider processor and process as equivalent. In a synchronous application, all the processes are synchronized at regular points. A barrier is inserted (by the programmer or by a parallel compiler) at the point, in each process, where it must wait. All processes can continue from this point only when all the other processes (or a subset of them) have reached it.

The **master-slave barrier** *(MS)* is shown in Fig. 3(a) for a configuration of 4 processes. It uses a centralized approach where a master process is responsible to lock and to release slave processes. Since barriers can be used more than once in a process, the MS algorithm must avoid that a process enters the barrier for a second time before previous processes have left the barrier for the first time. In order to avoid that, it uses a two phases mechanism: (i)*Arrival Phase:* Every process enters this phase and does not leave it until all processes have arrived at the same phase, (ii)*Departure Phase:* When every process finishes the Arrival Phase, a released command is given by the master.

The **tree barrier** *(Tr)* is shown in Figure 3(c) for a configuration of 8 processes. Similarly to the MS barrier, this barrier needs arrival and departure phases. Each phase is divided into $log_2 N$ stages (where N is the process count). For the example shown in Figure 3(c), there are three stages. In the first stage: P1 sends message to P0; P3 sends message to P2; P5 sends message to P4; P7 sends message to P6; (when P1,P3,P5,P7 reach their barriers). In the second stage: P2 sends message to P0; P6 sends message to P4. In the third stage: P4 sends message to P0 (P0 terminates arrival phase). Departure phase is based on a reverse tree construction.

The **butterfly barrier** *(Bf)* is shown in Figure 3(d) for 8 processes. Differently from the previous barriers, this barrier does not need any two phase protocol since it synchronizes pairs of processes at each stage (as before $log_2 N$ stages). In the example, we have in the first stage: P0 sends to P1, P2



(a) Router Architecture (b) Mesh (c) Torus

(d) Ring (e) Clustered-ring (f) Fat-tree

**Figure 2: The router architecture (with the equivalent gates count for different number of bidirectional ports) and the five selected topologies where each router shows the number of bidirectional ports.**

sends to P3, P4 sends to P5, P6 sends to P7. In the second stage: P0 sends to P2, P1 sends to P3, P4 sends to P6, P5 sends to P7. In the third stage: P0 sends to P4, P1 sends to P5, P2 sends to P6, P3 sends to P7.

The **all-to-all barrier** *(A2A)* is shown in Figure 3(b) for 4 processes. Differently from the previous barriers, it has only one phase and one stage making it the fastest barrier in the selected set. In the example, every process can simultaneously send a message to the other processes. However, the number of messages involved in this implementation grows quadratically with the number of processes.

Assuming that each core (where the process is running) is able to send at most one message at a time we can calculate the theoretical latency of each barrier algorithm based on the number of stages and phases. Table 2 summarizes the number of phases, the stages in each phase, the number of messages sent and the theoretical latency for each barrier implementation as a function of the number of processes. In the rest of the paper, we indicate the different barriers as $A2A$ for all-to-all, $MS$ for Master-Slave, $Bf$ for Butterfly and $Tr$ for Tree.

## 4.1 Barrier Implementations

The four selected barrier algorithms have been implemented in a SW library by using the message passing interface layer presented in Section 3(based on *send()* and *recv()* primitives). Figure 4 shows the SW implementation of the $MS$ barrier (similar implementations have been done for the other barriers).

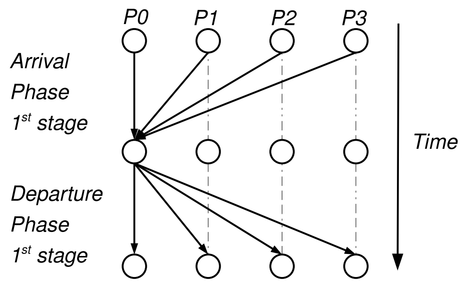
**Table 2: Phases, stages in each phase, number of messages sent and theoretical latency of each barrier as function of the number (N) of processes. The value of $k_i$ is dependent on the barrier implementation $i$.**

| Barrier | Phases | Stages x phase | Number of Messages |
|---|---|---|---|
| A2A | 1 | 1 | $N \times (N-1)$ |
| MS | 2 | 1 | $2 \times (N-1)$ |
| Bf | 1 | $\log_2 N$ | $N \times \log_2 N$ |
| Tr | 2 | $\log_2 N$ | $2 \times \log_2 N \times (N-1)$ |

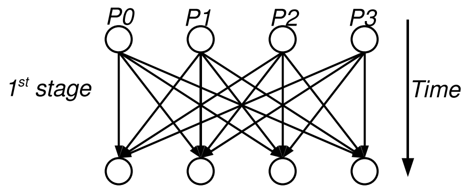
| Barrier | Theoretical Latency |
|---|---|
| A2A | $k_{A2A} \times N$ |
| MS | $2 \times k_{MS} \times N$ |
| Bf | $k_{Bf} \times \log_2 N$ |
| Tr | $2 \times k_{Tr} \times \log_2 N$ |

For the HW implementations, we used a custom design, placed in the communication interface of each core. The basic blocks of our HW implementation are a *Barrier State Buffer (BSB)* and a *Barrier Logic Unit (BLU)*. A schematic view of the communication interface with barrier HW support is given in Fig. 5.

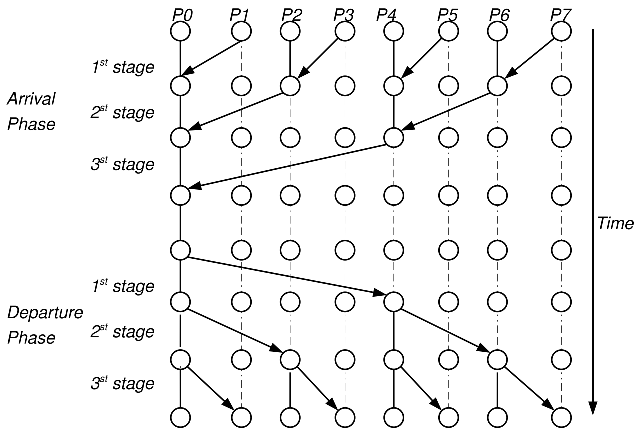
The HW barrier call (i.e. *barrier()*), from the SW layer, adds a barrier command to the *output buffer*. The command is trapped by the BLU which starts to execute the barrier



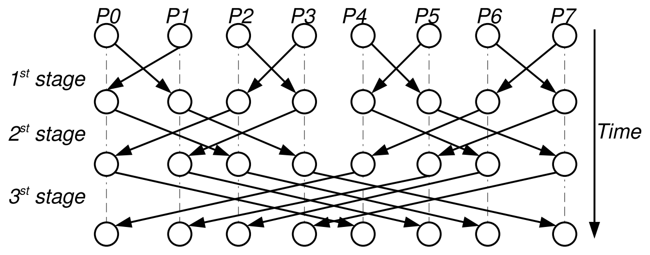
(a) Master-slave



(b) All-to-All



(c) Tree barrier



(d) Butterfly barrier

**Figure 3: Master-slave and All-to-All barriers with 4 processes; Tree barrier and Butterfly barriers with 8 processes.**

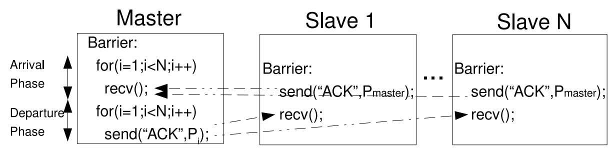


**Figure 4: SW implementation of the master-slave barrier, based on the message passing interfaces.**

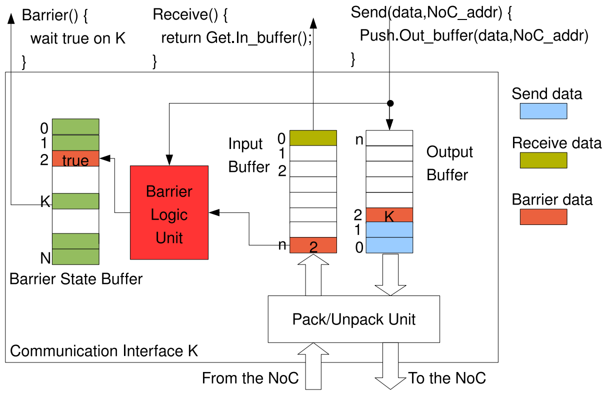


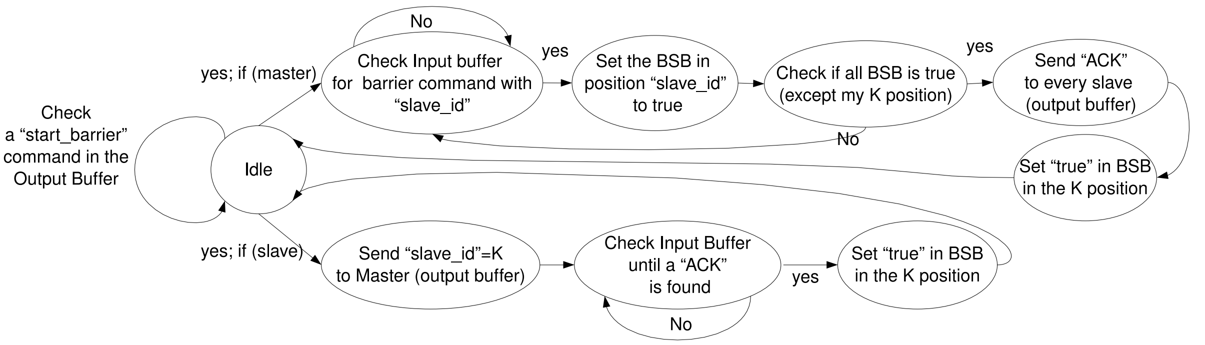
**Figure 5: Communication Interface with HW message passing and barrier support. On top the application layer commands used to access to the device, on the bottom the pack/unpack unit to communicate over the NoC.**

algorithm. The HW barrier call of the K processor waits for a "true" on the "K" location (locally spinning or using a sleep/wake-up mechanisms) of the *BSB*. The *BSB* has a number of entries equal to the number of involved processes. It is in charge of the *BLU* to set "true" the value in the "K" position when the barrier algorithm is completed. In the meanwhile, data coming from the other processes in the *input buffer* are trapped by the *BLU* and added to the *BSB* in order to be processed from the barrier algorithm implemented. In Fig. 6 we show the state machine implemented in the BLU for the MS barrier (similar implementations have been realized for the other barriers).

## 5. EXPERIMENTAL SETUP

Our experiments have been carried out with a cycle accurate SystemC [1] simulator. We designed it starting from *ppc750sim*[2], a SystemC simulator, which has a claimed accuracy of 10-15% on SPEC CINT 2000 compared to a real PowerPC 750 microprocessor. We added to the model the cycle-accurate SystemC description of our memory mapped communication interface (which implements the HW message passing and barrier layer). The router architecture as well as the different topologies have also been modelled with SystemC. Finally we used a single clock signal for the overall

[2]Available at *http://microlib.org*



No
yes; if (master)
Check Input buffer for barrier command with "slave_id"
yes
Set the BSB in position "slave_id" to true
Check if all BSB is true (except my K position)
yes
Send "ACK" to every slave (output buffer)
No
Check a "start_barrier" command in the Output Buffer
Idle
Set "true" in BSB in the K position
yes; if (slave)
Send "slave_id"=K to Master (output buffer)
Check Input Buffer until a "ACK" is found
yes
Set "true" in BSB in the K position
No

**Figure 6: FSM of the Barrier Logic Unit (BLU) for the master-slave barrier.**

system. Our experiments consist of measuring the completion time (as number of clock cycles) of the different barriers while varying system configurations. Before entering and after exiting a barrier, each cores randomly accesses the external memory, with an average injection/ejection in the NoC of 1 Byte/cycle. This ensures a fair analysis of the NoC behavior under "normal" traffic conditions. We carefully decided the injection/ejection rate as a good balance between the 1 Byte/Flops metric normally used in scientific computing [13] in presence of a balanced memory hierarchy, and the fact that the PPC750 is able to retire 2 instructions per clock (single precision floating point). We measured barrier completion time by using simulator cycle counters, starting them when processors enter a barrier and stopping them when they exit. The final completion time is the average of the completion times for the all processes. We compiled the barrier libraries as well as the main code with gcc-4.2.0 using optimization flag "-O3".

## 6. EXPERIMENTAL RESULTS

### 6.1 Scalability of barriers for different NoC topologies

Our first goal is to analyze the *scalability* of the different barriers (A2A=All-to-All, MS=Master-Slave, Bf=Butterfly, Tr=Tree) by varying the NoC topologies (Mesh, Torus, Ring, Clustered-Ring, Fat-Tree) as the number of cores(N) increases (see Fig. 7). We refer to the topology size only by the number of cores, however the topology takes also into account the EMIs (External Memory Interfaces) as reported in Table 1.

For the SW version of the barriers, we report the scalability only for the Mesh topology (see Figure 7(a)), since for the other topologies the SW barriers behave similarly in absolute terms. This consideration is also supported by the results reported in Figure 8 discussed in Section 6.2.

Analyzing Figure 7(a), we can observe that the SW implementations of the *A2A* and *MS* are linearly dependent on N as theoretically expected (see Table 2). From linear regression, we estimated a dependency[3] of $\approx 74 \times N$ and $\approx 48 \times N$ for *A2A-SW* and *MS-SW* respectively. This trend corresponds to a ratio of 3:1 between $k_{A2A-SW}$ and $k_{MS-SW}$. This ratio is due to the overhead of propagating all the messages involved in the A2A algorithm from the HW up to the SW layer and vice-versa.

[3]Calculated with a linear regression from data in Fig. 7(a). Regression error less than 1%.

From Figure 7(a), we can also observe that the SW implementations of *Bf* and *Tr* follow the logarithmic trend of the latency as given in Table 2. The logarithmic dependency[4] for *Bf-SW* and *Tr-SW* are $\approx 170 \times \log_2 N$ and $\approx 90 \times \log_2 N$ respectively, corresponding to a ratio of 1:1 between $k_{Bf-SW}$ and $k_{Tr-SW}$. Globally, *Bf-SW* results always the fastest SW barrier, making it the ideal candidate to perform barrier synchronization regardless system size and NoC topology when no HW support is provided.

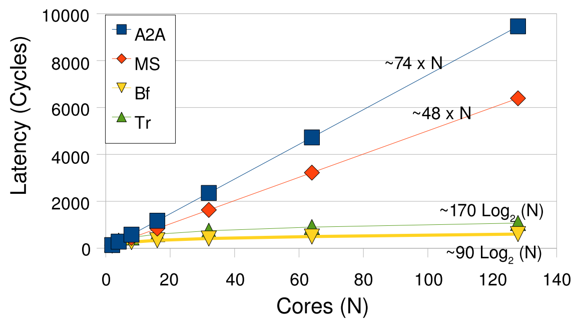
From Figures 7(b) to 7(f), we can see that also all HW barriers follow the theoretical trends shown in Table 2. The effects of the different NoC topologies on barrier performance can be quantified by linear regression in terms of the $k_i$ coefficients.

Comparing Figure 7(a) with Figures 7(b) to 7(f) we can see that, although the *A2A* barrier is the slowest SW implementation, the corresponding *A2A* HW implementation is one of the fastest, due to its parallel nature (see Figure 3). Except for the Ring topology, where the *A2A-HW* barrier outperforms the other HW-barrier implementations independently of the number of cores, for the other topologies there is always a crossing point between the *A2A-HW* and the *Bf-HW*. More in detail, these crossing points for Mesh, Torus, Clustered-Ring and Fat-Tree correspond to 32, 64, 80 and 128 cores respectively. The Ring is a particular topology for which the locality of the communication required by the barrier algorithm is disrupted. For this reason, sophisticated barrier algorithms cannot take advantage of Ring topology and the single-phase brute force of the *A2A* approach is always the winning solution making it the ideal candidate for this network topology.

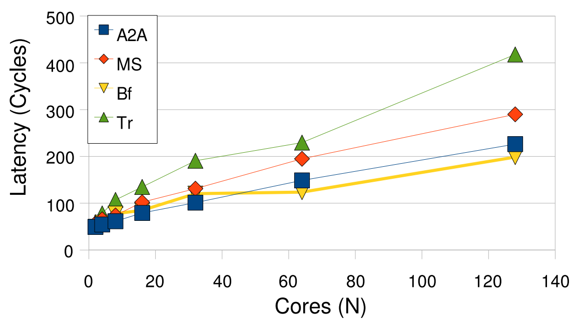
An opposite SW versus HW behavior can be noted for the Tree barrier (*Tr*). In fact, comparing Figure 7(a) with Figures 7(b) to 7(f), we can see that the *Tr-SW* is the fastest SW barrier, however when implemented in HW it is always the slowest barrier regardless the NoC topology. The reason for this behavior is that *Tr* is composed of $2 \times \log_2 N$ stages, while all other barriers have less stages (see Figure 3 and Table 2). In this sense, the parallel nature of the communication over NoCs prefers algorithms with a smaller number of stages than a smaller number of total exchanged messages.

Comparing *A2A-HW* to *MS-HW*, the HW behavior is different than those noted for the SW: *A2A-HW* is faster than

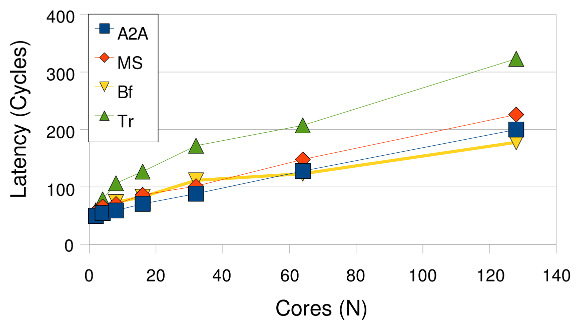
[4]Calculated with a linear regression from data in Fig. 7(a). Regression error less than 3%.



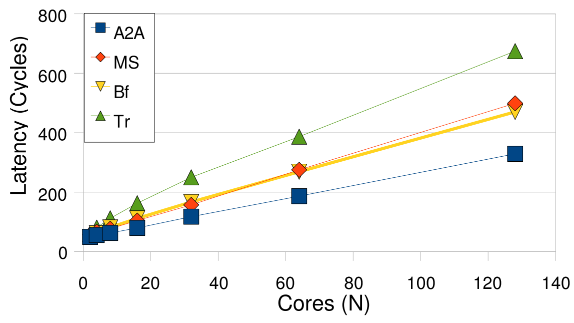
(a) SW barriers on Mesh



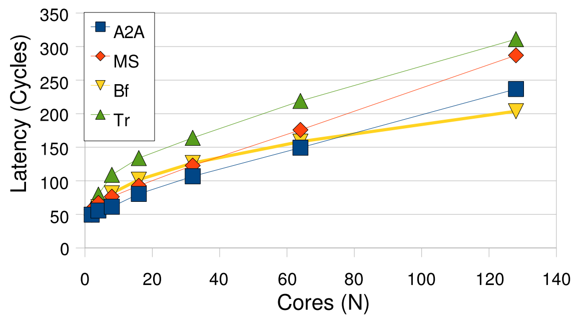
(b) HW barriers on Mesh



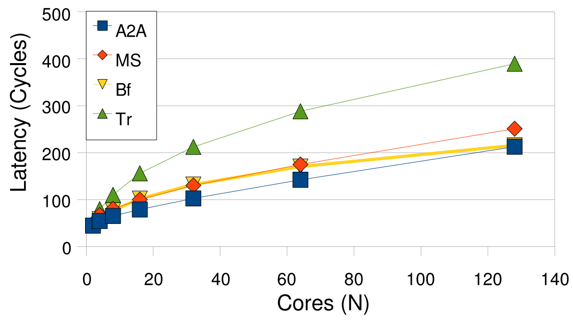
(c) HW barriers on Torus



(d) HW barriers on Ring



(e) HW barriers on Clustered-Ring



(f) HW barriers on Fat-Tree

**Figure 7: Scalability of the different SW barrier for the Mesh topology and of the HW barrier for the five topologies as the number of cores (N) increases (4, 8, 16, 32, 64, 128).**

*MS-HW*, while we found that *A2A-SW* is slower than *MS-SW*. This is due to the fact that, in the HW implementation, the incoming messages of the *A2A* algorithm are elaborated within the communication interface of each core and only the final result of the barrier is propagated to the SW layer.

## 6.2 Barriers overhead for different NoC topologies

The objective of this Section is to analyze the impact of different Noc topologies on the performance of the selected HW/SW barriers. In particular we will show the barriers overhead for different NoC topologies with respect to an *ideal channel* which provides zero latency for every network access. Figure 8 shows the overhead on the HW/SW barriers for different NoC topologies with respect to the *ideal channel* for systems of 16, 32, 64 and 128 cores.

We can see that the topology does not influence the *A2A-SW* and *MS-SW* barriers, and it has a small impact on *Bf-SW* and *Tr-SW*. This confirms that the rate at which the SW implementations inject data on the NoC (even relying on fast HW message passing layer) is so small that any topology could easily accomplish the task. In other words, if we are using message passing SW barriers we cannot expect to see any substantial difference in barrier latency by adopting an expensive highly connected NoC (e.g. Torus) with respect to a cheap NoC (e.g. Ring) (see Table 1 for a cost comparison). For instance, considering 128 cores and the *Tr-SW* barrier (the SW barrier with the highest NoC dependent behavior), in the worst case, the barrier performance decreases of 49% going from an *ideal channel* to a ring configuration. Considering a clustered-ring (the second cheapest NoC topology), the performance decreases is limited to 16%. A different scenario characterizes HW im-

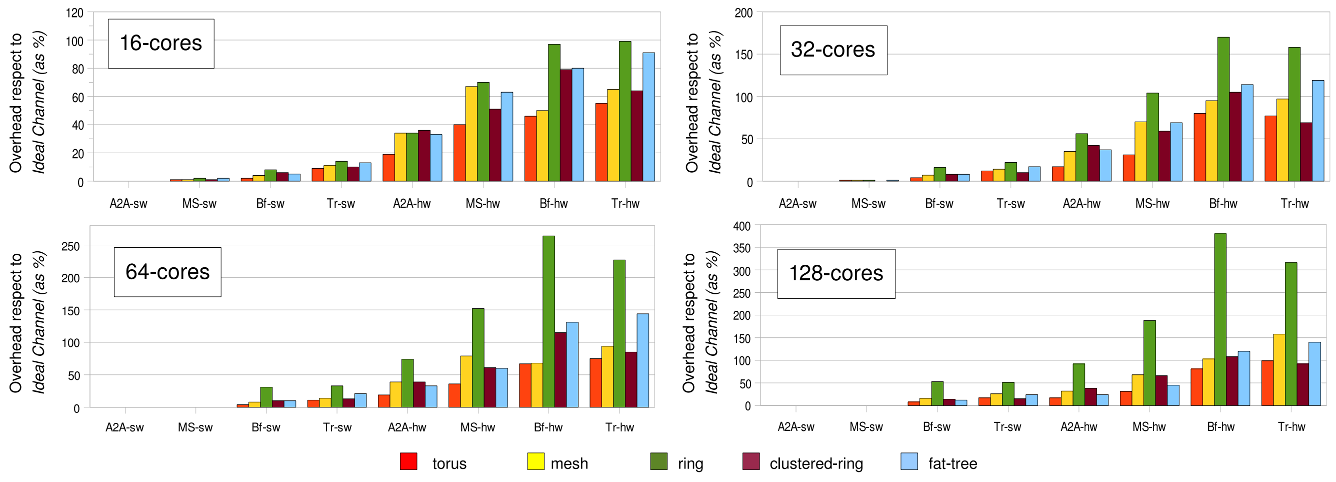


**Figure 8: Overhead of the different barriers for the different topologies respect to an *Ideal Channel*, for system composed of 16, 32, 64 and 128 cores**

plementations. Due to the fast HW protocol, NoC topology has a more relevant impact. Highly connected Torus topologies always provide a reduced overhead with respect to other topologies (except for Clustered-Ring combined with Tr-HW for 32 and 128 cores). The performance overhead introduced by Ring topology is the highest for the considered configurations. Noticeable is the performance obtained by Clustered-Ring, which is very close to the one obtained by Mesh topology, even if its cost is much smaller.

### 6.3 Latency/Complexity trade-offs of HW barriers combined with NoC topologies

As final result, we analyze the latency/complexity trade-offs of HW barriers combined with different NoC topologies. Figure 9 shows the Pareto points in terms of barrier latency and NoC complexity cost "C" (shown in Table 1) by varying the number of cores. Oblique dashed lines have been added to separate the system sizes from 4 to 128 cores. Among the Pareto points corresponding to the same number of cores, we identified as the most efficient combinations (corresponding to the points closer to the origin) the pair Clustered-Ring/*A2A-HW* for system of size up to 64 cores, while the pair Clustered-Ring/*Bf-HW* for systems of 128 cores.

## 7. CONCLUSIONS AND FUTURE WORKS

In this paper, we tried to quantitatively answer to the following questions: Could NoCs represent a low latency channel for barriers or vice-versa can barriers take advantage of the low latency and parallel nature of the NoCs? Is it correct to believe that more we invest in a NoC, more efficiently we can perform barrier synchronization? Our answer is that SW barriers (even relying on HW message passing layer) have so much overhead to under utilize any NoC configuration and that only HW barriers exhibit a relevant topology dependent behavior capable to fully exploit NoCs. Regardless NoC topology and system size, the butterfly algorithm is the best SW approach due to the small number of exchanged messages that must be propagated and processed in the SW layer. On the other hand, HW barriers with a small number of stages outperform barriers with small number of exchanged messages for medium-scale systems (32-64 cores) in almost every NoC topology we studied. However, as the number of cores increases, there is a number N of cores (which differs according to the topology) where it is more convenient to perform barrier synchronization with a low number of total exchanged messages. Moreover, the complexity of a NoC topology does not always pay in terms of efficiency, and "a simple" topology such as the Clustered-Ring can obtain very interesting results while performing barrier synchronization. The reason is that, especially when cores count grows, communication requirements for collective operations are very different from the requirements imposed by large-grain, cache-line-size data movements. We forecast that the results presented in this paper can be extended to other collective communication operations which we are currently studying to extend our work.

## 8. REFERENCES

[1] *SystemC 2.0 User's Guide*[5].

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] L. Benini and G. D. Micheli. Networks on Chip: A new SoC paradigm. *IEEE Computer*, 35(1):70–78, January 2002.

[4] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys*, 38(1), 2006.

[5] Cavium. Octeon plus cn58xx multi-core mips64. Available at: `http://www.cavium.com/OCTEONPlus_CN58XX.html`.

[6] W. J. Dally and B. Towles. Route packets, net wires: on-chip inteconnectoin networks. In *DAC'01: Proceedings of the 38th conference on Design automation*, pages 684–689, New York, NY, USA, 2001. ACM Press.

[7] IBM. PowerPC 750 RISC microprocessor technical summary. Available at: `http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/750_ts.pdf`, January 1998.

[8] Intel. From a few cores to many: A tera-scale computing research overview. Available at:

[5]Available at www.systemc.org

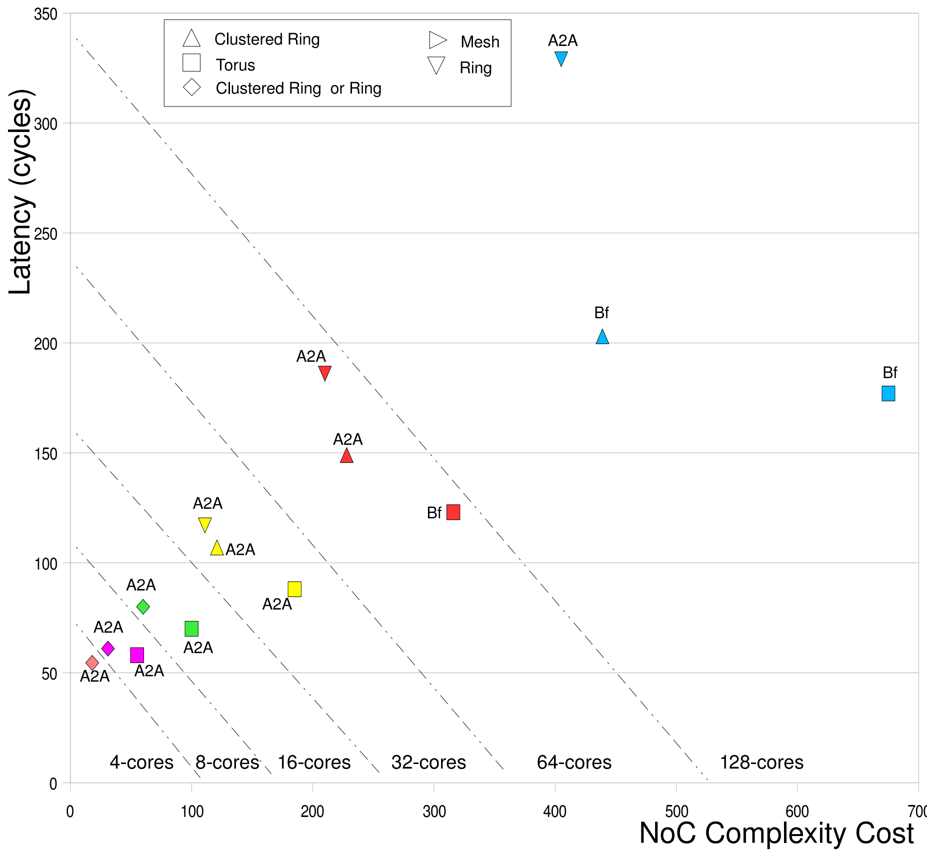


**Figure 9: Pareto points in terms of HW-barrier latency and NoC complexity cost by varying the number of cores from 4 to 128.**

`ftp://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf`.

[9] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin. Exploiting barriers to optimize power consumption of cmps. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 5.1, Washington, DC, USA, 2005. IEEE Computer Society.

[10] A. Marongiu, L. Benini, and M. Kandemir. Lightweight barrier-based parallelization support for non-cache-coherent mpsoc platforms. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 145–149, New York, NY, USA, 2007. ACM.

[11] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. Efficient synchronization for embedded on-chip multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(10):1049–1062, October 2006.

[12] R. D. Mullins, A. F. West, and S. W. Moore. Low-latency virtual-channel routers for on-chip networks. In *Proc. of the 31st Annual Intl. Symp. on Computer Architecture (ISCA)*, pages 188–197, 2004.

[13] R. Murphy, A. Rodrigues, P. Kogge, and K. Underwood. The implications of working set analysis on supercomputing memory hierarchy design. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 332–340, New York, NY, USA, 2005. ACM.

[14] G. Palermo and C. Silvano. PIRATE: A framework for power/performance exploration of network-on-chip architectures. In *PATMOS-04: Proceedings of International Workshop on Power and Timing Modeling, Optimization and Simulation*, September 2004.

[15] M. Palesi, R. Holsmark, S. Kumar, and V. Catania. A methodology for design of application specific deadlock-free routing algorithms for NoC systems. In *Proc. Intl. Conf. on Hardware-Software Codesign and System Synthesis*, Seoul, Korea, Oct. 2006.

[16] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie. Hardware- and software-based collective communication on the quadrics network. In *NCA '01: Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01)*, page 24, Washington, DC, USA, 2001. IEEE Computer Society.

[17] Tilera. Tile64 processor family. Available at: `http://www.tilera.com/pdf/ProBrief_Tile64.pdf`.

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

[19] W. Yu, D. Buntinas, R. L. Graham, and D. K. Panda. Efficient and scalable barrier over quadrics and myrinet with a new nic-based collective message passing protocol. *ipdps*, 09:182b, 2004.